Material for Hardware Experiments

Course C

Department of Electrical Engineering and Computer Science

Modified on June 3rd, 2020

Modified on June 8th, 2020

Content

Chapter 1 Introduction

1.1 Purpose and outline of experiment

This experiment aims to design and manufacture the CPU with simple architecture, as a final goal. To acquire the necessary knowledge and skill for that, the following issues will be learned.

1. Design and implementation of a simple digital logical circuit
2. Relation between computer architecture and logical circuit/electronic circuit
3. Design of logical circuit using CAD
4. Operation check using FPGA (Field Programmable Gate Array)

The CPU used in this experiment is the 32 bits CPU (KAPPA3-RV32I) with the level of architecture and instruction set, which is needed to learn the basic structure and function of computer. In this experiment, the operation of the CPU is grasped using the board mounted with FPGA, then the logical circuit performing the same operation is designed by each person using the hardware description language. And the designed circuit is actually mounted on the FPGA board to inspect and check the operation. Also, prior to these works, a very simple digital circuit is to be designed and manufactured at first to master the basic skill on the design of logical circuit.

1.2 Schedule of experiment

This experiment is conducted basically following the schedule below. A report should be submitted at the end of each theme.

* Hardware experiment 1

1. Design of logical circuit (7 classes)
   Design the logical circuit as follows, using CAD tools of the educational computer.
   - Several logical circuits are to be designed using the hardware description language (Verilog-HDL).
   - The logical circuits designed by the hardware description language are to be logically synthesized using CAD, and actually operated by being downloaded to the FPGA board.

* Hardware experiment 2

1. Understanding the operation of CPU (3 classes)
   Download a CPU to be actually designed to the FPGA board and check the operation. Pay attention to the data flow of each phase and timing of each module's operation.
2. Design and operation check of CPU circuit (about 12 classes)
   Design each module using Verilog-HDL. As a guideline of the design is explained for each module, the design is made by each person following the guideline. Download the designed circuit to the FPGA and check whether it operates as specified.

*** PBL-II**

1. **Development of program on the manufactured CPU (3 classes)**
   Develop the program which operates on the manufactured CPU, using C language. Also describe the program including the memory mapped I/O and interruption.
2. **PBL (6 classes)**
   Create games and demonstrations utilizing the input and LED, by the program operating on the manufactured CPU. The achievements are to be presented on the final day.

1.3 Notice on report writing

Regardless of this experiment, there are required conditions that experiment reports should meet. Paying attention to the conditions, write and submit the report.

1. **Strictly keep the deadline for submission.**
   Submit your report without fail. You can't earn the credits if any one of reports is lacking. Pay attention that some credits related to this experiment are the requirement for starting the graduation thesis. Also, the deadline for submission should be kept without fail. The report missing the deadline will be marked low even if the content is excellent. Furthermore, in the case that the submission is much delayed, the report can't be marked and the credit may be disapproved as it is.

2. **Make it clear what the purpose is, what kind of the experiment is done, what result is obtained, what question is answered and what consideration is made thereby.**
   The report should be written so that a third person can clearly understand the issues above. Here, the third person is not a teacher, but a person who doesn't have any information on this curriculum though the person has general knowledge on the hardware. The report, in which only the obtained results are written without any explanation, only the calculation results are written without mentioning the question to be answered, and no discussion is stated, will be given the lowest rating. If the rating through all the experiments is insufficient, the credit may not be given.

3. **Write the discussion for the answer to the question. Even an elementary school student can make the calculation for a question only to obtain a numerical value and consult a book only to copy the content. What is required here is what consideration is made in answering the question.**

4. **Give a title and description to figures, graphs and tables. It is totally unclear the meaning of a figure without a title and a graph without the description on what each axis expresses. The title**

and necessary information should be described for figures, tables and graphs.

5. **Try to differentiate your report from the others. It is important to ask others what you can't understand. However, it is crucial on such occasions to understand what was taught and to be able to explain it by your own words. Even if the obtained result was the same or you belonged to the same group, it is meaningless to submit the report that was copied straight from others.**

1.4 Composition of this guidebook

The composition of each chapter is as follows.

Chapter 2 Logical design using Verilog-HDL
Explanation on the work of Hardware Experiment 1 and the exercises in the report.
Chapter 3 Design of micro-processor
Explanation on the work of Hardware Experiment 2 and the exercises in the report.
Chapter 4 Assembly development on KAPPA3-RV32I
Software creation on the developed microprocessor
Appendix A Reference to Verilog-HDL syntax
Appendix B Description of FPGA board MU500-RXSET01 for experiment
Appendix C Specifications for micro-processor KAPPA3-RV32I for education

Chapter 2 Logical design using Verilog-HDL

2.1 Outline of experiment

Regarding the following exercises 1 ~ 5, describe the circuit using Verilog-HDL. Then, after making the logic synthesis using the CAD tool (Quartus), download the design data to the FPGA board and check the operation.

The Verilog-HDL files for the circuits that have already been designed are placed in the public folder. As the skeleton (an empty file) of Verilog-HDL file for the design object is placed in the templates, make use of it.

2.1.1 How to use the Quartus

In this experiment, the development tool for FPGA, Quartus, is used. The brief description on how to use the Quartus is given in the following. Also refer to the operation procedure document on FPGA design tools, docs/MU500-manual/MU500-RXSET01/1 users manual/2. In the Quartus, the design data is managed by the unit called "project". Then, create the project following the procedure below.

1. **Select the menu of 'File -> New Project Wizard'.**
2. **Input the following information to the next screen.**
   - **The place storing the project's data.**
   - **Project name**
   - **The name of top level module of Verilog-HDL**

   Though it is not a strict rule, an individual folder (directory) should be prepared for each project to reduce troubles. The place of this directory may be different from that of Verilog-HDL's files. In the case that the designated folder doesn't exist, it is automatically created. Also, the name of the top level module should be the same as that of the project, to reduce mistakes. As all of the top level modules have been prepared for this experiment, make use of those names. For example, if the project name is 'Exercise 1', the top module's name is 'kadai1'.

3. **Files to be used can be added in the next screen, but you may click 'Next' without designating any files, because they can be added later. In the case that the files have been given in advance, they may be added here.**
4. **The FPGA devices to be used are designated in the next screen. Here, select the 'EP4CE30F2317' and click 'Next'. To narrow down the list at the center, input the information such as 'Device family', 'Package' and 'Pin count'.**

5. **The external tools to be used is registrated in the next screen, but do nothing here and click ‘Next’.**
6. **Click ‘Finish’ in the confirmation screen.**
7. **The process for project’s creation has been completed.**
8. **Select the menu of ‘Assignments -> Import Assignments...’, and input ‘verilog-src/templates/topmodule.qsf’ into the dialog. This process connects the I/O port of the Verilog-HDL’s top level module to various I/O pins of the FPGA board.**
9. **After finishing the creation of Verilog-HDL files, select the menu of ‘Project -> Add/Remove files in project....’ and make the addition of files.**
10. **Make the compilation by clicking the ▷ button on the upper tool bar.**
11. **Unless errors are output in the lower message log (Errors are output in the red.), the compilation is done successfully. Even if there are warnings indicated in blue, the FPGA data are produced normally. However, the cause of failure may be implied there, so refer to the warnings in the time of debugging.**
12. **After connecting the FPGA board to a USB port of the PC and turning the power on, select the menu of ‘Tools -> Programmer’, and press the start button in the dialog which appears on the screen. At that time, make sure that the ‘USB-Blaster[USB-0]’ is displayed at the upper-left column of ‘Hardware Setup …’. In the case that the writing was done correctly, the upper-right progress bar turns green and it should be displayed ‘100%(Successful)’ therein. In the case of failure, confirm the connection and try again.**
13. **In the case that the files were modified, repeat the procedure from the 10th process. In the case of adding files, return to the 9th process.**
14. **In the case of starting the work in the existing project, select the menu of ‘Files -> Open project...’ and make the addition or modification of files.**
15. **In the case of editing files in the Quartus, select the menu of ‘view -> Project navigator’ and turn the type of ‘Project navigator’ to be ‘Files’. Then the list of files used in the project appears on the screen. By clicking the file name, the content of the file will appear at the center.**
16. **The creation and modification of files can be made using generic text editors, but the character code of the files should be UTF-8 in the case that Japanese language is included in comments.**

2.2 Exercise 1

Using the Verilog-HDL, describe the combinational circuit that outputs the 8 bits signal, 1 bit of which is used for a dot, to display numerals in the ‘7 segment LED’, regarding the 4 bits signal as a 4 digits binary number (2’b0000 – 2’b1111: 0 – 15) .
Refer to Figure B.3 regarding the ‘7 segment LED pattern’.

Temperate for the decode circuit for the 7 segment display

```
module decode_7seg(input [3:0] in,
                        output [7:0] out);
...
endmodule
```

[Hint] Make use of the function statement and case statement.

After finishing the design of the decode_7seg, create the project 'kadai1' with 'led driver.v' as an additional file and compile it. Regarding the pin assignment, import the 'topmodule.qsf'.

Confirm that the '7SEG-LED' is correctly displayed according to the value of rotary switch.

2.3 Exercise 2

Describe the 4 bits adder using the Verilog-HDL.

Template for the 4 bits adder

```
module add4(input [3:0] a, b, // 4 bits input x 2
            input cin, // Carry input
            output [3:0] s, // 4 bits output
            output cout); // Carry output
...
endmodule
```

[Hint] The corresponding description is given in Figure A.2. If possible, not using that as it is, create the description only using the '+' operator and concatenating operator. The point is to regard the 'carry output' as the sum of the fifth bit.

After finishing the design of the 'add4', create the project 'kadai2' by adding the files of the 'kadai2.v', 'led_driver' and 'decode_7seg.v', and compile it using the Quartus.

Make sure that the display is correctly made according to the two rotary switches and the push button at the upper-right corner.

2.4 Exercise 3

Design the 4 bits up/down counter.

Template for the 4 bits up/down counter

```
module udcount4(input clock, // Clock signal
                input reset, // Reset signal
                input ud, // 0: Up, 1: Down
                input enable, // Count enable signal
                output [3:0] q, // Output of count value
                output carry); // Carry output
...
endmodule
```

[Operation] This is the counter to increase or decrease the value of count one by one, basically following the 'clock'. The value of count is initialized when the 'reset' becomes zero. The count is up when the 'ud' is zero and down when the 'ud' is one, while it is not altered when the 'enable' is zero.

Also, the 'carry' is turned to be one, when transiting from 4'b1111 to 4'b0000 in the up-mode and transiting from 4'b0000 to 4'b1111 in the down-mode.

[Hint] The always statement, if statement and non-blocking assignment statement are used. However, the carry signal is to be created by the combinational circuit, using the assign statement.

After finishing the design of the 'udcount4', create the project 'kadai3' to which the 'kadai3.v', 'led driver' and 'decode_7seg.v' are added, and compile it using the Quartus.

Confirm that the display is correctly made according to the rotary switch and the push button at the upper-right corner. In particular, make sure whether the carry output is generated correctly. For that, the clock frequency is needed to be slowed to some extent. Use D or E of the clock's rotary switch.

[Challenging exercise] If you are afford to do, design the 1 digit decimal (0 - 9) up/down counter by altering the udcount4.

2.5 Exercise 4

Design the ‘key input encoder’ and ‘key input buffer’.

Temperate for the ‘key input encoder’

```
module keyenc(input [15:0] keys,
                            output key_in,
                             output [3:0] key_val);
...
endmodule
```

The keys are the 16 bits input signal, that are connected with the push switches of the FPGA board, psw a0 - psw a3, psw b0 - psw b3, psw c0 - psw c3 and psw d0 - psw d3, via the synchronizer (described later). It is not necessary to take the synchronizer into consideration in the design of the ‘keyenc’. The outputs are the 4 bits signal, ‘key_val’, and the 1 bit signal, ‘key_in’. When either of push switches is pressed, the ‘key_in’ becomes one and the value corresponding to the number pressed is output to the key_val. In the case the ‘key_in’ is zero (the push switch is not pressed), the ‘key_val’ can be any value. Don’t care that in this case.

The lay-out of the push switches is shown in Figure 2.1. Here, the values of the push switches in the right- side row, ‘Clear’, ‘+’, ‘-’ and ‘=’, are not used in the ‘keyenc’. (In the firat place, they are not included in the keys.)

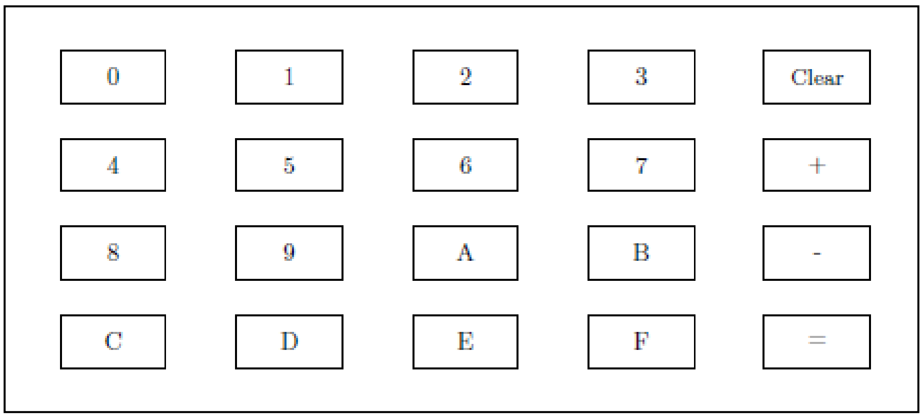


Figure 2.1 Lay-out of push switches

[Hint] The ‘keyenc’ is the perfect combinational circuit. But it is complicated to describe using logical expression, so the if statement or casex statement is used in the function statement. In the case of the casex statement, create the label of the casex statement, taking into consideration the possibility that plural push switches are pressed at the same time.

Template for the 'key input buffer'

```
module keybuf(input clock,
              input reset,
              input key_in,
              input [3:0] key_val,
              input clear,
              output [31:0] out);
...
endmodule
```

The 'key input buffer' is the register retaining the values of 32 bits, whose input is the 'key_in' notifying the input and the 'key_val' representing the key value at the time. When a new value is input (key in = = 1), the previous value is shifted to left (to the upper side) by 4 bits and the new value is entered in the lower 4 bits. For example, in the case that the previous value is 32' h0123 4567 and the 'key_val = 4' h8' is newly input, the previous value is shifted by 4 bits, turning to be 32' h1234 5670. And as the value of the 'key_val' is entered in the lower 4 bits, it turns to be 32' h1234 5678. When the 'clear' is one, the value retained in the 'input buffer' is turned to be 32' b0.

After finishing the design of the keyenc and keybuf, create the project, kadai4, by bringing together the files of kadai4.v, led_driver.v and decode_7seg.v, and compile it by the Quartus.

[Synchronizer]

Template for the synchronizer

```
module syncro(input clock,
              input reset,
              input in,
              output out);
...
endmodule
```

The synchronizer is the circuit to synchronize the signal of the push SW that is pressed asynchronously with the clock, with the clock signal. Further, the event of pressing the push SW is converted into the pulse for one clock, by the synchronizer. Use the synchronizer that is given in advance.

2.6 Exercise 5

Design of simple calculator

A simple calculator is designed by combining the circuit modules created in the exercises so far.

Temperate for the calculater

```
module calc(input clock,
            input reset,
            input [15:0] keys,
            input clear,
            input plus,
            input minus,
            input equal,
            output [31:0] ibuf,
            output [31:0] cbuf);
...
endmodule
```

The calculator has a register retaining the 32 bits calculation results inside (buffer for the calculation results). It is operated as follows.

- **All are turned to be zero at the time of resetting.**
- **Upon pressing the clear key, the 'calculation results buffer' and 'input buffer' are cleared.**
- **Upon pressing the ten key, numeric numbers are entered into the 'input buffer'. (Use the keyenc and keybuf.)**
- **Upon pressing '+' (plus) key, the existing value of the 'key input buffer' is transferred to the 'calculation buffer', and the 'key input buffer' is turned to be zero.**
- **Upon pressing the '=' (equal) key, the sum of the existing value of the 'key input buffer' and the value of the 'calculation result buffer' is substituted into the 'calculation result buffer', and the 'key input buffer' is turned to be zero.**

The description on the top module using 'calc' is given in the kadai5.v. The key and LED are assigned as follows.

- **The input values are displayed in eight '7SEG-LEDs' of MU500-RK.**
- **The 'psw a4' is assigned to the 'clear' key.**
- **The 'psw b4' is assigned to the '+' key.**
- **The 'psw d4' is assigned to the '=' key.**
- **The H group of the MU500-7SEG (8 pieces) are used to display the calculation results.**

[Challenging exercise] If you have time, improve the calculator so that it can make the subtraction besides the addition. In that case, it is necessary to memorize whether the previous operation was '+' or '-', when the '=" is pressed.

Also, how should it be modified, in order to calculate and display not as hexadecimal number but as decimal number ? Discuss it in the report.

2.7 Exercises of the first report

1. On the exercises 1～5, describe the issues below.

- Specifications for the circuit to be designed

- Description on the design using Verilog-HDL

- Brief explanation on circuit description

Also, answer to the challenging exercises, if you are afford to do.

2. In the case of not using the method describing circuit by Verilog-HDL, it is necessary to make the design by the following procedure.

- State number minimization of finite state machine

- State encoding of finite state machine

- Creation of the truth table for state transition function and output function

- Creation of the Karnaugh diagram for state transition function and output function

- Simplification of the production-sum-type logical equation for state transition function and output function

- Creation of the logical circuit from the production-sum-type logical equation

Compared to this procedure, consider the merits and demerits of the method performing the logical synthesis from the Verilog-HDL description. Particularly, are there any points that the design procedure above is thought to be lead ?

3. Describe your thoughts and comments on this experiment.

End of Chapter 2

Chapter 3 Design of microprocessor

3.1 Overall schedule

The schedule is shown in Table 3.1. The content of work is only a guide, so it may be moved up or moved down.

Table 3.1 Schedule

	Content
1st Class	Outline of how to use the board
2nd Class	Understanding of operation
3rd Class	Questions
4th Class	Manufacture of KAPPA3-LIGHT
5th Class	(Manufacture of stconv and ldconv)
6th Class	
7th Class	(Manufacture of phasegen)
8th Class	
9th Class	(Operation check of debugger module)
10th Class	
11th Class	(Manufacture of controller)
12th Class	
13th Class	
14th Class	(Operation check)
15th Class	

3.2 Work procedure

3.2.1 Outline of how to use the board and understanding of operation

- Reading the reference materials, understand the internal structure, operation and control of KAPPA3-RV32I (KAPPA3-LIGHT).
- Download samples of design data to the FPGA board.
- Convert the sample program 1 to the machine language.
- Input the converted machine language to the memory on the FPGA.
- Operate the KAPPA3.
- Understand the operation of program.

- Observe how the internal registers change in each phase of respective instructions.
- Considering how control signals change in each phase of respective instructions, create a table like the Table 3.2.
- There are some control signals that can work whatever their values may be. (don't care) For those cases, enter the '-'.

Table 3.2 Table showing controlsignals

Instruction	Phase	PC_SEL	PC_LD	MEM_SEL	MEM_READ	MEM_WRITE	MEM_WRBITS	IR_LD	RD_SEL	RD_LD	A_LD	B_LD	A_SEL	B_SEL	ALU_CTL	C_LD
Lui x5, 12345h	IF															
	DE															
	EX															
	WB															
Addi x5, x5, 678h	IF															
	DE															

How to generate machine language

Here, we explain how to generate the machine language, taking lui x5 and 12345h as examples. The lui is a U-type instruction, [31:12] is an immediate value, [11:7] is r_d and [6:0] is an operand code of 7'b0110111. Because the r_d is x5, it is entered 5'b00101 in [11:7]. The immediate value is 12345h, so it is converted to 20'b0001_0010_0011_0100_0101 in binary code (Verilog-HDL notation). Mind that one digit in hexadecimal code corresponds to four digits in binary code. They are connected to be 32'b0001_0010_0011_0100_0101_0010_1011_0111. Pay attention that the delimiter for the r_d field and operand code doesn't coincide with that of four digits. Finally, converting this binary code into the hexadecimal code, it turns to be 32'h1234_52B7 (123452B7h in the assembly notation). Because values are input as hexadecimal number in KAPPA3-LIGHT, enter this 123452B7h. (The last 'h' is not entered.)

How to input the program

1. Set the DIP-A[0] of the FPGA board to be the input mode.
2. Set the HEX_A of the FPGA board to be 8(MAR).
3. Enter the head address (10000000) of the program. Make sure that the MAR's display becomes 10000000.
4. Set the HEX_A of the FPGA board to be 9(memory).

5. **Enter the machine language. Make sure that by entering 123452b7, for example, the Mdr' display becomes 123452b7.**
6. **Press the '+' button of the FPGA board. The MAR's value is added by 4.**
7. **Return to 5.**

In the case of failing the input, alter the MAR's value, or alter the address by the '+' or '-' button and overwrite a correct value.

3.2.2 Questions

- **Convert the sample program 2 into the machine language.**
- **Enter the converted machine language to the memory on the FPGA.**
- **Operate the KAPPA3.**
- **Understand the operation of the program.**
- **Observe how the internal registers change at each phase of respective instructions, and create a table after the Table 3.2.**
- **Consider how various control signals change at each phase of respective instructions.**
- **Report the results of observation and consideration to instructors.**

3.2.3 Manufacture of KAPPA3-LIGHT

Manufacture the KAPPA3-LIGHT, following the specification document. Specifically, it is designed in the procedure below.

1. **Production of ldconv and stconv**
   Complete the description of the content, in reference to the templates/ldconv.v and templates/stconv.v.
2. **Production of phasegen**
   Complete the description of the content, in reference to the templates/phasegen.v. Create the project with public/phasegen_test.v as a top module. And add the designed hasegen.v, public/led_driver.v and public/syncro.v, then compile it. Confirm that the phase transition is correctly made according to the push SW. As for the clock SW, Use C or D.
3. **Confirmation of debugger module**
   Create the project with the public/kappa3_light.v as a top module, and add the public/debugger.v, public/led_driver.v, public/kappa3 light_core_dp.v public/mem64kd.v, public/memory.v, public/reg32.v, public/regfile.v and public/syncro.v, and the keyenc.c and

keybuf.v designed in the Hardware Experiment 1 to the project. Then, compile it. Though the public/kappa3_light_core_dp.v is a module including all register memories of KAPPA3-LIGHT, it doesn't operate as a processor because it has no control circuit. Here, make sure that values can be written into each register memory via the debugger module, and they can be displayed.

4. Production of kappa3_light_core

Copy the public/kappa3_light_core_dp.v on an appropriate folder, in the name of 'kappa3_light_core.v'. Editing this file, produce the kappa3_light_core. The files to be added to the project are as follows.

- public/kappa3 light.v
- public/led driver.v
- public/debugger.v
- public/alu.v
- public/mem64kd.v
- public/memory.v
- public/reg32.v
- public/regfile.v
- public/syncro.v
- keyenc.v --- Prepared in Hardware Experiment 1
- keybuf.v --- Prepared in Hardware Experiment 1
- ldconv.v --- Prepared in Chapter 1
- stconv.v --- Prepared in Chapter 1
- phasegen.v --- Prepared in Chapter 2
- controller.v --- To be produced here based on templates/controller.v
- kappa3 light core.v--- To be produced here based on public/kappa3_light_core_dp.v

Specifically, instantiate the stconv, ldconv and alu in reference to Figure C.2, and make the connection of each register memory. Further, constitute the control circuit to output control signal line, with the phasegen and controller. Then instantiate it.

5. Completing all the procedure, download the design data to the FPGA board. Check the operation of each instruction following Table 3.3. The program should be input all to the address of 10000000h. The rows that are written 'machine language' give the instructions to be input. Setting the memory address to be '10000000h' in the input mode, enter the values directly. As described in the next row that is written 'pre-condition', enter the values of each register and memory in the input mode. After that, switching to the execution mode, execute one instruction only (STEP_INST). Check whether the values in the registers and memories equal to the content of the row of 'post-condition'. Also, though it is not written explicitly, set appropriate values for the register memory of the targets to be checked, so that the values should become different from those to be checked in the 'post-condition'. For example, even if the register's value that was zero

originally becomes zero again after executing an instruction, it will be unclear whether the value turned to zero as the execution's result or it is not changed at all. This case, set the value other than zero, like FFFFFFFFh, beforehand.

3.3 Sample Program

3.3.1 Sample program 1

```
10000000   lui     x5,   12345h
10000004   addi    x5,   x5,   678h
10000008   lui,    x6,   98765h
1000000C   addi    x6,   x6,   432h
10000010   add     x7,   x5,   x6
10000014   jal     x0,   -4h
```

Notice : In representing a hexadecimal value in a program, the 'h' is attached at the end of the expression. For example, since '15h' is '15' in the hexadecimal number, it equals to '21' in the decimal number. The '-4h' becomes 'FFFFFFFCh' in the 2's complement representation of 32 bits.

3.3.2 Sample program 2

```
10000000    lui    x5,    10000h
10000004    addi x5,    x5, 100h
10000008    lw    x6,    x5, 000h
1000000C    lw    x7,    x5, 004h
10000010    blt    x6,    x7, 004h
10000014    sw    x5,    x7, 008h
10000018    jal    x0,    004h
1000001C    sw    x5,    x6, 008h
10000020    jal    x0,    -4h
....
10000100 12345678h
10000104 9ABCDEF0h
```

- What is this program doing ?

- What happens if the instruction in the address of 10000010 is rewritten ‘bltu’.

3.3.3 Check list

Table 3.3 : Check list for the confirmation of operation

Mnemonic	Machine language	Pre-condition	Post-condition

LUI x01, 12345000h	123450B7	pc = 10000000	pc = 10000004
			x1 = 12345000

AUIPC x02, 86543000h	86543117	pc = 10000000	pc = 10000004
			x2 = 96543004

JAL x03, -100h	F01FF1EF	pc = 10000000	pc = 0FFFFF04
			x3 = 10000004

JALR x04, x05, 100h	10028267	pc = 10000000
		x5 = 10000000
			pc = 10000100
			x4 = 10000004

BEQ x05, x06, 100h	10628063	pc = 10000000
		x5 = 0000000A
		x6 = 0000000A
			pc = 10000104

BEQ x07, x08, -100h	F08380E3	pc = 10000000
		x7 = 0000000A
		x8 = 0000000A
			pc = 0FFFFF04

BEQ x09, x10, 100h	10A48063	pc = 10000000
		x9 = 0000000A
		x10 = 00000000
			pc = 10000004

BLT x11, x12, 100h	10C5C063	pc = 10000000
		x11 = 0000000A
		x12 = 00000064
			pc = 10000104

BLT x13, x14, -100h	F0E6C0E3	pc = 10000000
		x13 = FFFFFFF6
		x14 = 00000064
			pc = 0FFFFF04

BLT x15, x16, 100h	1107C063	pc = 10000000
		x15 = 00000064
		x16 = 0000000A
			pc = 10000004

BLTU x17, x18, 100h	1128E063	pc = 10000000
		x17 = 0000000A
		x18 = 00000064
			pc = 10000104

BLTU x19, x20, 100h	1149E063	pc = 10000000
		x19 = FFFFFFF6
		x20 = 00000064
			pc = 10000004

continued to the next page

continuation from the previous page

Mnemonic	Machine language	Pre-condition	Post-condition

LB x05, x21, 0h	000A8283	pc = 10000000
		x21 = 10000100
		M[10000100] = 000000A5
			pc = 10000004
			x5 = FFFFFFA5

LB x06, x22, 9h	009B0303	pc = 10000000
		x22 = 100000F8
		M[10000100] = 0000A500
			pc = 10000004
			x6 = FFFFFFA5

LB x07, x23, 6h	006B8383	pc = 10000000
		x23 = 100000FC
		M[10000100] = 00A50000
			pc = 10000004
			x7 = FFFFFFA5

LB x08, x24, -1h	FFFC0403	pc = 10000000
		x24 = 10000104
		M[10000100] = A5000000
			pc = 10000004
			x8 = FFFFFFA5

LW x09, x25, -4h	FFCCA483	pc = 10000000
		x25 = 10000104
		M[10000100] = 12345678
			pc = 10000004
			x9 = 12345678

SB x26, x27, 0h	01BD0023	pc = 10000000
		x26 = 10000100
		x27 = 0000005A
		M[10000100] = 00000000
			pc = 10000004
			M[10000100] = 0000005A

SB x28, x29, 9h	01DE04A3	pc = 10000000
		x28 = 100000F8
		x29 = 0000005A
		M[10000100] = 00000000
			pc = 10000004
			M[10000100] = 00005A00

SB x30, x31, 6h	01FF0323	pc = 10000000
		x30 = 100000FC
		x31 = 0000005A
		M[10000100] = 00000000
			pc = 10000004
			M[10000100] = 005A0000

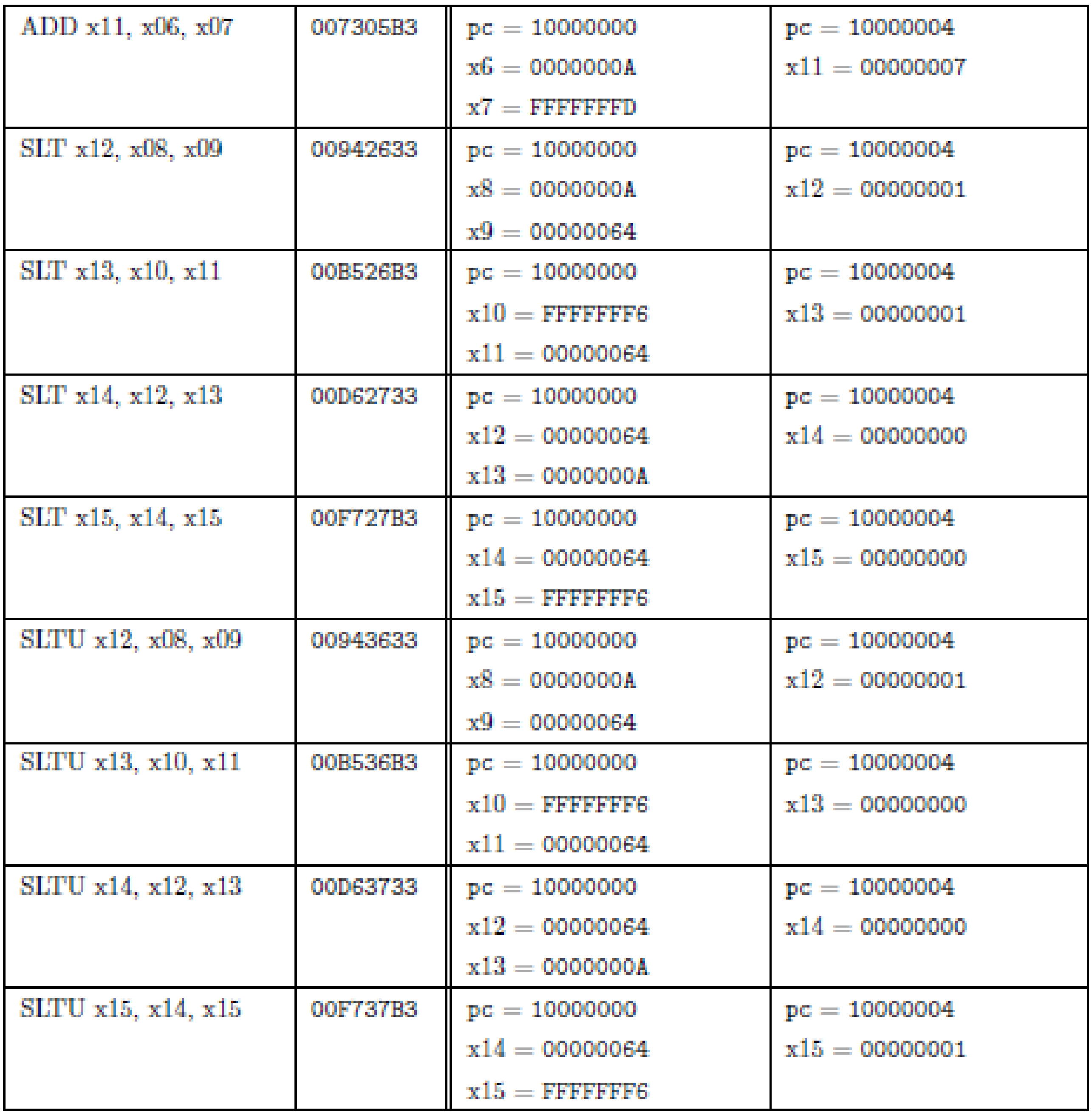
SB x01, x02, -1h	FE208FA3	pc = 10000000
		x1 = 10000104
		x2 = 0000005A
		M[10000100] = 00000000
			pc = 10000004
			M[10000100] = 5A000000

SW x03, x04, -4h	FE41AE23	pc = 10000000
		x3 = 10000104
		x4 = 5AA500FF
			pc = 10000004
			M[10000100] = 5AA500FF

ADDI x10, x05, -3h	FFD28513	pc = 10000000
		x5 = 0000000A
			pc = 10000004
			x10 = 00000007

continued to the next page

continuation from the previous page



Mnemonic	Machine language	Pre-condition	Post-condition

ADD x11, x06, x07	007305B3	pc = 10000000
		x6 = 0000000A
		x7 = FFFFFFFD
			pc = 10000004
			x11 = 00000007

SLT x12, x08, x09	00942633	pc = 10000000
		x8 = 0000000A
		x9 = 00000064
			pc = 10000004
			x12 = 00000001

SLT x13, x10, x11	00B526B3	pc = 10000000
		x10 = FFFFFFF6
		x11 = 00000064
			pc = 10000004
			x13 = 00000001

SLT x14, x12, x13	00D62733	pc = 10000000
		x12 = 00000064
		x13 = 0000000A
			pc = 10000004
			x14 = 00000000

SLT x15, x14, x15	00F727B3	pc = 10000000
		x14 = 00000064
		x15 = FFFFFFF6
			pc = 10000004
			x15 = 00000000

SLTU x12, x08, x09	00943633	pc = 10000000
		x8 = 0000000A
		x9 = 00000064
			pc = 10000004
			x12 = 00000001

SLTU x13, x10, x11	00B536B3	pc = 10000000
		x10 = FFFFFFF6
		x11 = 00000064
			pc = 10000004
			x13 = 00000000

SLTU x14, x12, x13	00D63733	pc = 10000000
		x12 = 00000064
		x13 = 0000000A
			pc = 10000004
			x14 = 00000000

SLTU x15, x14, x15	00F737B3	pc = 10000000
		x14 = 00000064
		x15 = FFFFFFF6
			pc = 10000004
			x15 = 00000001

End of Table 3.3

3.4 Exercises of the second report

1. **Consider pipelining this processor. Then, because the instruction fetch and the access to data memory can't br executed at the same time, as they are, the instruction memory and data memory need to be separated. (Avoidance of structure hazard) Besides that, what cases should be considered ? Also, describe specifically what should we do, to solve those problems.**

2. **Describe your thoughts and comments on this experiment.**

Chapter 4 Assembly development on KAPPA3-RV32I

4.1 Overall time schedule

The schedule is shown in Table 4.1. This content of work is only a guide, so it may be moved up or moved down.

Table 4.1 Schedule

	Content
1st Class	**Input of C language, compile, simulation and actual machine execution**
2nd Class	**Memory-mapped I/O and bit manipulation**
3rd Class	**Timer interrupt**

4.2 hello world by assembly

4.2.1 Input assembly

Firstly, download sample programs and header files from the GitHub to the D drive. Then, the file of ‘sample_7seg.py’ is created in ‘D:¥hw2019¥asm’ that was cloned before, so look into it.

sample1_7seg.py: Display ‘1’

```
program.append( Inst.LUI(5, 0x04000000) )    # Substitute the address of 7-segment into r5
program.append( Inst.ADDI(10, 0, 0x60) )      # Substitute the pattern of 7-segment ”1” into r10
program.append( Inst.SB(5, 10, 0x00) )      # r5[0] = 0x60 (Store 0x60 to the address of 7-segment)
program.append( Inst.JAL(0, -4*2) )      # Unconditional branch to the previous instruction
```

This is the program to display ‘1’ in the seven-segment LED. The detail of the program is to be understood later.

4.2.2 Assemble

Boot the Anaconda terminal and move to the cloned d drive.

```
d:
cd hw2019¥asm
```

Next, assemble it.

```
python sample1_7seg.py
```

Upon execution, the input program is assembled and displayed.

```
sample program 1
040002b7 | LUI x05, 4000000h
0ff00513 | ADDI x10, x00, ffh
00a28023 | SB x05, x10, 0h
ff9ff06f | JAL x00, -8h
```

Also, what the assembled result is converted to the intel hex format is stored in the file of the sample1_7seg.hex. The file name of the output destination is designated by the “file name = 'sample1 7seg.hex' ” in the sample1_7seg.py. Hereafter, in the case of copying and using this file, do not forget to change the name of the output file.

```
:04000000040002B73F
:040001000FC0051314
:0400020000A2A82395
:04000300FF9FF06FFC
:00000001FF
```

4.2.3 Download to a real machine

Booting the Quartus, download the kappa3_rv32i.sof in rv32i-asm, by ‘tools → programmer’. This is what the RISC-V RV32 is implemented.

After finishing the download, boot the ‘in-system memory content editor’ from the menu of ‘tools’. As another window opens, the content of the RAM inside the FPGA can be altered by selecting the ‘USB-Blaster’ in the upper-right ‘Hardware:’. In order to load the sample1.hex produced above, open the context menu by right-clicking on the window mentioned above. Next, by selecting the sample1.hex in the ‘Import Data from File …’, the loading is carried out. Further, by opening the context menu by right-clicking again and selecting the ‘Write Data to In-system Memory’, this content is transferred to the memory on the FPGA.

After transferring to the memory, reset the system. Then, the program is executed.

4.3 Input/output

4.3.1 Outline

Though it is not necessary to be conscious in the development of program on PC, it is necessary to understand the mechanism of input/output (I/O) to some extent in the program directly handling hardware like this class. The I/O is what the software operating on the processor directly receives

the input from outside and sends the signal outside. For example, that corresponds to what the software detects a button pressing and turns on LEDs.

There are some methods to realize the I/O. In this experiment, we use the memory-mapped I/O, where I/O equipment is controlled in the same way as an ordinary access to a memory. Regarding how the processor, and even the software, accesses to memories, refer to the lecture materials in Computer Architecture 1.

On the I/O equipment, a small-scale memory called register is loaded. Pay attention that this register is different from that in the processor. In the case of the memory-mapped I/O, the group of registers can be treated in the same way as an ordinary memory. Namely, if there is a special variable to access a register, the substitution to the variable corresponds with that to the register, and the reference to the variable corresponds with that to the register. (Because the address assignment for an ordinary variable can't be set (or is difficult to be set), the variable to access the register is defined using a pointer.)

4.3.2 Memory-map

The memory-map in this experiment environment is shown in the table blow. The memory-map is the correspondence table indicating which equipment is connected to an address range in the memory space. The software related to the I/O is described following this table. Details of each equipment are explained in the next sections.

Start	End	Content
0x0200 0000	0003	CLINT (Timer)
0x0400 0000	004b	MU500 (LED)
0x1000 0000	ffff	RAM (64kbyte)

4.3.3. Seven-segment LED and dot matrix LED

- Outline The offset of the seven-segment LED and dot matrix LED is shown in the table. Th offset is an information on where the register is placed in the address range shown in the memory-map. Because the MU500 is placed in 0x0400 0000, the '0x0400 0000 + offset' is the address to be accessed actually. For example, the register to control the upper-leftmost seven-segment LED is placed in 0x0400 0000, because the address is '0x0400 0000 + offset 0'.

Start	End	Content	
0x00	0x3f	7-segment	8 lines x 8 rows x 8bit
0x40	0x47	Dot matrix	8 lines x 8 rows x 1bit

- Seven-segment LED A one byte register is assigned to each one of seven-segment LEDs. Regarding the content in one byte, refer to B.

For example, in the case that the shape of '0' is to be displayed, write '1111 1100' in binary code and '0xFC' in hexadecimal code. For example, describe the following.

```
program.append( Inst.LUI(5, 0x04000000
program.append( Inst.ADDI(10, 0, 0xFC) ) # "0" is displayed in 7-segment.
program.append( Inst.SB(5, 10, 0x00) )
program.append( Inst.JAL(0, -4*2) )
```

Because in this program the '0xFC' is substituted to the offset 0 of the address where the 7-segment of MU500 is placed, '0' is displayed at the upper-leftmost seven-segment LED.

- Dot matrix LED The dot matrix LED takes one byte by putting eight ones together. The left-uppermost eight ones are the offset '0x40', the left side is '0x41', the left side on the second line is '0x42' and so on. The memory assignment for the dot matrix LED is shown in Table 4.2.

Table 4.2 Memory assignment for dot matrix LED

Offset	MSB(7)	6	5	4	3	2	1	LSB(0)
0x40	A7	A6	A5	A4	A3	A2	A1	A0
0x41	A15	A14	A13	A12	A11	A10	A9	A8
0x42	B7	B6	B5	B4	B3	B2	B1	B0
0x43	B15	B14	B13	B12	B11	B10	B9	B8
0x44	C7	C6	C5	C4	C3	C2	C1	C0
0x45	C15	C14	C13	C12	C11	C10	C9	C8
0x46	D7	D6	D5	D4	D3	D2	D1	D0
0x47	D15	D14	D13	D12	D11	D10	D9	D8

Because A0 is the leftmost and A1 and A2 are the next in the actual assignment, the order is reversed in the case of writing the MSB in the left side. For example, in the case that the left-uppermost is to be flashed in turn such as light-on, off, on, off and so on, write 0101 0101 in binary code and 0x55 in hexadecimal code, so it is described as follows.

```
program.append( Inst.LUI(5, 0x04000000) )
program.append( Inst.ADDI(6, 0, 0x55) )
program.append( Inst.SB(5, 6, 0x040) )
```

4.3.4 Getting input

The offset related to the input is shown in the table below. One bit is assigned to each of buttons and DIP switches, so eight of those constitute one byte. One rotary switch is assigned one byte.

Start	End	Content	
0x48	0x4b	Button	4 lines X 5 rows
0x4c	0x4d	Rotary switch	8bit x 2
0x4e	0x4f	DIP	8bit x 2

There are 5 buttons in one line, so in total 20 buttons in four lines. Delimiting the lines at a good stopping point, express the content for one line in one byte. The memory assignment for push SWs is shown in Table 4.3. The upper 3 bits of each byte are not used. For example, in order to know the state of the upper-left push SW (A0), check the value of the lowest bit in the offset 0x48. The method of controlling each bit is explained later. The sample code to read the upper-left button is shown below.

Table 4.3 Memory assignment for push SWs

Offset	MSB(7)	6	5	4	3	2	1	LSB(0)
0x48	–	–	–	A4	A3	A2	A1	A0
0x49	–	–	–	B4	B3	B2	B1	B0
0x4a	–	–	–	C4	C3	C2	C1	C0
0x4b	–	–	–	D4	D3	D2	D1	D0

```
program.append( Inst.JAL(0, -28) )
program.append( Inst.LUI(5, 0x04000000) )
program.append( Inst.ADDI(6, 0, 0xFF) )
program.append( Inst.LW(10, 5, 0x48) )          # Get the value in the offset 0x48.
program.append( Inst.ANDI(11, 10, 0x01) )       # Check whether LSB is 1 or not, using ANDI.
program.append( Inst.BEQ(11, 0, 0x08) )         # Branch if the ANDI's result is 0.
# Process for the case that the button is pressed.
program.append( Inst.JAL(0, 4) )                # Branch to the end of the process.
# Process for the case that the button is not pressed.
program.append( Inst.JAL(0, -28) )      # Branch to the start of the program.
```

The ANDI instruction is used, in order to get the value of the offset 0x48 in the LW(10, 5, 0x48) and judge whether the LSB is 1 or not.

It is shown below the sample where the seven-segment LED is turned on in response to the button pressing.

The sample where the seven-segment LED is turned on in response to the button

```
program.append( Inst.LUI(5, 0x04000000) )
program.append( Inst.ADDI(6, 0, 0xFF) )
program.append( Inst.LW(10, 5, 0x48) )
program.append( Inst.ANDI(11, 10, 0x01) )
program.append( Inst.BEQ(11, 0, 0x08) )
program.append( Inst.SB(5, 6, 0x000) )
program.append( Inst.JAL(0, 4) )
program.append( Inst.SB(5, 0, 0x000) )
program.append( Inst.JAL(0, -28) )
```

4.3.5 Technique for bit control

In RISC-V assembly, the process can be controlled only on a 1 byte-by-1 byte basis even in the smallest unit. However, in many cases of handling the I/O, it is necessary to control on a smaller one bit unit. Here, the method to control values in the unit of 1 bit is outlined.

- Judgement of 0/1 for a bit　　　Make a judgement on whether a bit is 0 or 1. For example, the judgement on whether the button is pressed. This case, usually create a constant where only the bit to be judged is 1 and the other bits are 0, and perform an AND operation on the constant and the target register.

For example, carry out the procedure below, in order to judge if the LSB is 1 or not.

1. **Create a constant where only the USB is 1. Because it is 0000 0001 in binary code, this is converted to the constant of 0x01 in hexadecimal code.**
2. **Perform the AND operation for the target variable and the created constant.**
3. **If the button is pressed, the operation result becomes non-zero (1 in this case). If not pressed, it becomes 0.**

- Clear a bit to zero　　　Make a bit zero, that is called 'clear'. This case, create the constant where only the bit to be cleared is 0 and the other bits are 1, and perform the AND operation for this constant and the target variable. However, because it is usually simpler to describe the constant where only the bit to be cleared is 1 and the other bits are 0, it is performed a NOT operation for such constants in many cases. For example, in the case that the LSB is to be set 0, the following procedure is carried out.

1. **Create the constant where only the LSB is 0 and the others are 1. If expressing in one byte, it is '1111 1110' in binary code, so this is converted to the constant of 0xFE in hexadecimal code.**
2. **Perform an AND operation on the 0xFE and the target register.**

- Set a bit to 1　　　Describe the constant where only the bit to be set is 1 and the others are 0, and perform an　OR operation for the constant and the target register.

4.3.6 Exercise 1

Using two digits of the seven-segment LED, create the counter to display how many times the button is pressed, in decimal code. The position of the seven-segment LED and button to be used may be designed freely.

1. **Prepare the program in which the value of one digit decimal number, designated in this program, is displayed in a seven-segment LED.**
2. **Prepare the counter for one digit decimal number. Upon pressing a button, it is added by +1 and displayed in the seven-segment LED using the program above. When the count becomes 10, it returns to 0.**
3. **Prepare the counter for two digits decimal number.**

4.4 Reading and writing in memory

In the case that data more than the number of registers are needed to be saved, it is necessary to write them into a memory or read them from a memory. This section outlines the reading and writing of memory.

4.4.1 Simple writing into memory

A simple program to write into memory is shown in the following. In this program, the value increasing by 1 with the elapse of time is written into the address of 0x10000100.

Writing into memory

```
program = [
        Inst.LUI(5, 0x10000000),  # memory = 0x10000000
        Inst.ADD(6, 0, 0),        # x6 = 0 (counter)
        Inst.SW(5, 6, 0x100),     # loop: memory[x5+0x100] = x6
        Inst.ADDI(6, 6, 1),       # x6++
        Inst.JAL(0, -4*3)         # goto loop
]_
```

As mentioned before, the memory is placed from 0x10000000 to 0x1000ffff. The instruction strings are also arranged in the memory, and they are placed from 0x10000000. The program has 5 instructions, so the addresses from 0x10000000 to 0x10000014 are used to place the instruction strings. The area to save data is needed to avoid the area where the instruction strings are written. In this program, the data are stored in comparatively small address in order to make it easy to see by the In-System Memory Content Editor, but the upper address should be used in real applications.

Executing the program, confirm whether the data are correctly written into the memory using the In-System Memory Content Editor. In the In-System Memory Content Editor, right-clicking the RAM to select the Read Data from In-System Memory, the value of the memory on the present FPGA can be written into the Editor. Here, mind that the address on the In-System Memory Content Editor is different from that designated in the program. The address on the In-System Content Editor is the offset address from 0x10000000, and it is a quarter (shifting right by 2 bits) of the address designated in the program. For example, the data is written into 0x10000100 this time, but the offset from 0x10000000 is 0x100 and it further becomes one quarter of that. Therefore, it looks like the data is written into 0x40, in the In-System Content Editor

4.4.2 Set a constant in memory and use it

The program to count-up one digit number displayed by seven-segment LED is shown in the following. In this program, the lighting pattern in seven-segment LED is stored in the memory address of 0x10001000 beforehand, and utilize it to turn on the 7-segment.

Set a constant into memory

```
program = [
          # Store 7seg led patterns
          Inst.LUI(5, 0x10001000),          # memory = 0x10001000
          Inst.ADDI(6, 0, 0b11111100),      # 7seg: 0
          Inst.SB(5, 6, 0x0),
          Inst.ADDI(6, 0, 0b01100000),      # 1
          Inst.SB(5, 6, 0x1),
          Inst.ADDI(6, 0, 0b11011010),      # 2
          Inst.SB(5, 6, 0x2),
          Inst.ADDI(6, 0, 0b11110010),      # 3
          Inst.SB(5, 6, 0x3),
          Inst.ADDI(6, 0, 0b01100110),      # 4
          Inst.SB(5, 6, 0x4),
          Inst.ADDI(6, 0, 0b10110110),      # 5
          Inst.SB(5, 6, 0x5),
          nst.ADDI(6, 0, 0b10111110),       # 6
          Inst.SB(5, 6, 0x6),
          Inst.ADDI(6, 0, 0b11100000),      # 7
          Inst.SB(5, 6, 0x7),
          Inst.ADDI(6, 0, 0b11111110),      # 8
          Inst.SB(5, 6, 0x8),
          Inst.ADDI(6, 0, 0b11110110),      # 9
          Inst.SB(5, 6, 0x9),
          # Main
          Inst.LUI(6, 0x04000000),
          Inst.ADD(7, 0, 0),           # x7 = 0 (counter)
          Inst.ADD(8, 7, 5),           # address of memory[x7]
          Inst.LB(9, 8, 0),            # x9 = memory[x7]
          Inst.SB(6, 9, 0),
          Inst.ADDI(7, 7, 1),           # x7++
          Inst.JAL(0, -4*5)             # goto loop
]_
```

he part where the count-up is performed.

The part where constants are placed in the memory stores 0b11111100 for the seven-segment LED's lighting pattern "0", into the offset 0 of 0x10001000 and stores 0b01100000 for the seven-segment LED's lighting pattern "1", into the offset 1, and so on. The part where the count-up is performed turns on the seven-segment LED according to the present count number stored in x7. For that, it is

necessary to calculate the memory address where the seven-segment LED's lighting pattern is stored, based on the present count. Then, calculate the '0x10001000 + x7'. Because the lighting pattern corresponding to the x7 is stored there, the pattern will be loaded. Store the loaded lighting pattern into the seven segment LED's register. (Note that once the count becomes 9, this sample program doesn't work correctly after that. How should the program be altered so as to work correctly ?)

4.5 Timer

In the case of treating time, it is necessary to use a timer. The program that reads the timer's present value and writes it into the memory of 0x10000100 (64bit) is shown below.

Read the timer

```
program = [
        Inst.LUI(5, 0x02000000),
        Inst.LUI(6, 0x10000000),
        Inst.LW(7, 5, 0x4),
        Inst.SW(6, 7, 0x104),
        Inst.LW(7, 5, 0x0),
        Inst.SW(6, 7, 0x100),
        Inst.JAL(0, -4*5)
]_
```

The clock number from the start of the system is stored in the mtime register. The mtime register's address is 0x02000000 and its size is 64 bits (8 bytes). This sample program reads the register word by word (4 bytes) and stores it to the memory of 0x10000100. Make sure that it is actually stored, using the In-System Memory Content Editor. As mentioned before, the apparent address in the In-System Memory Content Editor becomes 0x40.

Appendix A
Reference to Verilog-HDL syntax

The Verilog-HDL syntax is briefly described in this chapter. Confirm the details with references. There are two types of Verilog-HDL syntax, the original (Verilog1995) and the new one called Verilog2001. The Verilog2001 is fooward compatible with the original Verilif-HDL. Here, the original Verilog-HDL syntax is described. The Verilog2001 syntax is also explained partly. Both of the two types are acceptable in currently available tools (programs).

A.1 Outline

Though the Verilog-HDL seems a program language very similar to the C language, it has some features, as it is specialized for the description of hardware.

First, The descriptions on the Verilog-HDL is classified to the one to "synthesize" hardware and the one to "simulate" the hardware, While the description to synthesize can be used for the simulation, the description to simulate can't be necessarily used for the synthesis. Namely, the description to synthesize is written with the purpose of how to create a program. On the other hand, the description to simulate is written with the expectation of how a program is to be worked. In this experiment, the description on the synthesis is mainly explained. The description on simulaton is explained as needed.

Next, in the Verilog-HTL, available syntax elements differ depending on the place. This is what is difficult to understand, as compared to common program languages. The syntax elements are roughly classified as follows.

- **Declaration**
- **Item**
- **Statement**
- **Expression**

Among them, only the decralation and item can be described directly under the module (indicating whole of the circuits, detail mentioned later). The "statement" that can be described naturally is not describable actually. This is attributed to that the Verilog-HDL has been developed to describe hardware. Be careful about this point, as it is easy to make a mistake before getting used to it.

A.1.1Components (Item)

As mentioned above, only the Decralation and components (Item) can be described directly under

the module. There are various Items in Verilog-HDL. What is often used is explained here.

- assign statement

- function declaration

- always statement

- init statement

- instance description

Though it is written the assign statement and always statement, these are not Statements, but Items in the strict Verilog-HDL syntax. The assign statement indicates the wire connection in hardware. The function declaration is, in some sense, what specifications for combinational logic circuits are described in the form of function. However, the target circuit is not created only by the function decralation. The function is needed to be called in an Expression (explained later). The always statement is used for the description on the vent regularly repeated. This was originally used for the purpose of simulation, and it can also be used as the description for the synthesis for sequential circuit by limiting the writing method. The init statement is used to describe the events carried out only once. This is purely used for the simulation and not used for the purpose of synthesis.

A.1.2 Statement

The statements are the main components in the case of ordinary program languages, but in the Verilog-HDL they are describable only within the function, always and init. The main statements are shown below.

- if statement

- case and casex statement

- begin – end block

Besides these, a variety of statements are prepared for simulation, but all needed inthis experiment is to be able to use these statements. The details of each statement are described later.

A.2 Simple syntax rule

A.2.1 Identifier

The identifier is the name given to signal lines, terminals and modules, and it is a concept similar to the name of variables and functions in software programming. The identifiers in the Verilog-HDL are the string composed of alphabet (a…z, A...Z), numeral (0…9), underscore '_' and dollar '$', and the

initial character should be the alphabet. Also, a part of the strings are used as the word called reserved word that has a special meaning, so they can't be used as the identifiers. (For example, if and module.) Though the length of an identifier is prescribed to be within 1024 characters by the syntax rule, there are some cases that only the shorter lemgth can be identified depending on CAD tools. Further, as an upper-case letter is distinguished from a lower-case letter, 'signal' denotes a different identifier from 'Signal', but there is also some cases that they are not distinguished, depending on the CAD tools. In general, even if correct in syntax, it is unfavorable the description where 'signal' and 'Signal' are mixing up.

Example of identifiers

```
- Correct identifier
abc     clk     reset$   long_identifier   x1
- Incorrect identifier
99input  Numerical is at the head.
pin@mod1        Unavailable character @ is included.
or              Reserved word
```

Besides that, there is a rule that arbitrary strings beginning with '￥' (back-slash) are regarded as an identifier. However, this was introduced for the purpose like finding shelter in an emergency, that the data of another format is converted to the data of the Verilog-HDL. Therefore, this shouldn't be used as much as possible.

A2.2 Reserved word

The list of reserved words are shown in Table A.1. Because the strings described in the list have special meanings as the reserved words, they can't be used as idetifiers.

always	**and**	**assign**	**begin**	**buf**
bufif0	**bufif1**	**case**	**casex**	**casez**
cmos	**deassign**	**default**	**disable**	**edge**
else	**end**	**endcase**	**endfunction**	**endmodule**
endprimitive	**endspecify**	**endtable**	**endtask**	**event**
for	**force**	**forever**	**fork**	**function**
highz0	**highz1**	**if**	**innone**	**initial**
inout	**input**	**integer**	**join**	**large**
macromodule	**medium**	**module**	**nand**	**negedge**
nmos	**nor**	**not**	**notif0**	**notif1**
or	**output**	**parameter**	**pmos**	**posedge**
primitive	**pull0**	**pull1**	**pullup**	**pulldown**
rcmos	**real**	**realtime**	**reg**	**release**
repeat	**rnmos**	**rpmos**	**rtran**	**rtranif0**
rtranif1	**scalared**	**small**	**specify**	**specparam**
strong0	**strong1**	**supply0**	**supply1**	**table**
task	**time**	**tran**	**tranif0**	**tranif1**
tri	**tri0**	**tri1**	**triand**	**trior**
trireg	**vectored**	**wait**	**wand**	**weak0**
weak1	**while**	**wire**	**wor**	**xor**
xnor				

Table A.1 List of reserved words

A.2.3 Logical value

As a signal value, besides ordinary "1" and "0", the four values of "Z (or z)" and "X (or x)" are used. Z expresses the impedance (the state where all the transistors connected to the signal are off and the electric potential is not determined due to the high resistance value). X, the value that doesn't exist physically, is used to simulate the state where it is unknown which of 0 and 1 to be taken, like flip-flop's value just after supplying power, and also used as the value matching any pattern of 0 and 1 in casex statement (mentioned later).

A.2.4 Numerical value

Different from software programming languages, the numerical values in the Verilog-HDL are not only the values, but also have the information on the bit length. Also, the four codes of binary, octal,

decimal and hexadecimal are used for expressing numericals. Specifically, a numerical value in the Verilog-HDL is expressed in the format of
<bit length>' <cardinal number><no-signed numerical value>.
The symbols expressing the cardinal number are as follows.

B or b	**binary number**
O or o	**octal number**
D or d	**decimal number**
H or h	**hexadecimal number**

In the case of hexadecimal number, the numerics of 10~15 are expressed by A~F (or a~f). In the case that the bit length is omitted, it becomes 32 bits. Also, in tha case that the bit length and cardinal number are omotted, it becomes a 32 bits decimal number. In order to make a numeric with many digits easy to see, the '_' (underscore) can be inserted at an approoariate position. However, the compiler ignores only the underscore, so it is a desiner's responsibility to insert the underscore at the appropriate position like every 4 bits. Even if the underscores at the 4th bit and 8th bit are mistakenly inserted to the 3rd and 7th bit, the compliler ignores it.
Example of numerical value

1'b0	**0 of 1 bit**
4'b1001	**1001 in 4 bits binary (= 9 in decimal)**
5'b01XX	**5 bits bunary with the lower 2 bits undefined**
8'h5f	**5F in 8 bits hexadecimal (= 95 in decimal)**
156	**156 in 32 bits decimal**
16'b0001_0100_1010_1111	**Underscore's use case**

Because there is no conception of negative number in the Verilog-HDL, it is necessary for design-side to explicitly devise such means as using complement notation or preparing for sign bit , to handle the negative number.

A.2.5 Comment

The comment is a string that is described but ignored, and usually used for a memorundom or note so that the desiner can make the description easy to understand for one's own self and other people. In some cases, this is used to make the description invalidate without erasing it. There are two types of comment. One is the comment written from '//' to the end of the line, and the other is the comment written from '/*' to '+/'.
Example of comment

assign x = in; // Substitute in to x.	**Comment is written following '//'.**
/* this line is a comment	
this line is also a comment */	**Comment over plural lines**

A.3 Module description

The Verilog-HDL describes hardware using "module" as a basic unit. The meaningful description on the Verilog-HDL contains one module at least. The basic configuration of the module is shown below.
Configuration of module

```
module <Name of module> (<name of port>, <name of port>, …);
<Port declaration>
<Parameter decralation>
<Wire declaration>
<Reg decralation>

<Main body>
endmodule
```

The identifiers mentioned above are used for the names of the modules and the ports.With the turn indifferent from that of the port names here, the following port declaration can be made. In the Verilog-HDL, the statement usually ends with ';' (semi-colomn), but as it is clear the statement with the reservation word of the endXXXX group ends with the one word, the smi-colomn is not used. This case, the endmodule statement corresponds to that. Also, mind that on the contrary, the semi-colomn is needed at the end of the module statement.
The port declaration has the configuration below.
Port declaration

```
input   <Range designation > < signal name>, < signal name >, ... ; or
output  < Range designation > < signal name >, < signal name >, ... ; or
inout   < Range designation > < signal name >, < signal name, ... ;
```

The 'input' is a port declaration for input, the output is a port declaration for output and the inout is a port declaration for input/output. The range designation is written in the form of [<MSB >: <LSB >], and designate the positions of the signal line's uppermost bit (MSB) and lowest bit (LSB). In the case that the range designation is omitted, it is regarded as MSB = LSB = 0.
The wire decralation and reg declaration make the decralation of the signal line used inside the module. They comply with a common format and have the configuration as follows.
Wire declaration and reg declaration

```
wire    <Range declaration> <name of signal line>, < name of signal line >, ...;
reg     <Range declaration> <name of signal line>, < name of signal line >, ...;
```

What is noted is that when the reg declared signal is used for an external output, the output declaration should be made at the same time as the reg declration. However, as mentioned later, if the module declaration is made in the format of the Verilog2001, these declarations can be made together. Also, Because the signal used inside the module without the declarations of wire and reg is regarded not as an eror, but as the wire of one bit (the range designation of [0:0], this doesn't become an compile error in spite of wrong signal name due to the misdescription, but may cause other bugs.

The parameter declaration have the configuration as follows.
The Parameter declaration

```
parameter          <Identifier> = <numerical>;

Use case

parameter MEMSIZE = 256;
parameter BITWIDTH = 8;
.
.
.
reg [7:0]                    mem[0:MEMSIZE - 1];
wire [BITWIDTH:0]            syncro;
```

Various descriptions appear in the module's main body. The most often-used one in the synthesis are the assign statement, always statement and instance description.

A.3.1 Module declaration in Verilog2001

In addition to that mentioned abobe, modules can be declared in the following format in the Verilog2001.
Configuration of module in Verilog2001

```
module <Module name>(<port declaration>, <port declaration>, ...);
<Parameter declaration>
<wire decralation>
<reg declaration>

<Main body>
endmodule
```

In the original Verilog, only the port name was declared in the parentheses after the module name, and the declaration of the port itself was made inside the module. But in the Verilog2001, the port declaration can be made in the parentheses after the port name. It is desirable to use this format because the typing amount is usually reduced in the format. At the same time, it can be prevented the error that the identifier name in the port declaration is mistyped differently from the port name. To be exact, the parameter declaration can also be included in the parentheses after the module name, which is omitted here.

A.4 assign statement

The assign statement expresses the substitution of the value that is always defined.
assign statement

```
assign    <Name of signal line> = <equation>;
```

The left-side is usually the signal line's name itself by the wire declaration. In some cases, the statement with the range deisignation or concatenation operator is used as shown below.
Example of assign statement

```
wire [15:0]          bus;
wire [3:0]           a;
wire [3:0]           b;
wire                 carry;
wire [3:0]           sum;

assign bus[15:12] = a;                (1)
assign {carry, sum} = a + b;          (2)
```

In (1), the 4 bits signal line 'a' is connected to the 4 bits from the 12th bit to 15th bit in the signal line of bus. Such a notation as bus [15:12] is called 'range designation'. Originally, it is used when only a part of the signal line, 'bus', with the 16 bits width is targeted. The range designation can also be used in the left-side of the equation. Further, when the number indicating the range is 1, like bus[15], it expresses the signal line of 1 bit. In this example it is equivalent to bus[15:15].

In (2), the upper 1 bit (the 5th bit) in the result (5 bits) of addition of the 4 bits numbers is connected to 'carry', and the lower 4 bits is connected to 'sum'. Refer to the next section about the concatenation operation.

What is noted in the assign statement is that the signal line appearing in the left-side is the wire-decrared one. The reg-declared signal line can't be placed on the left-side. Also, the assign statement can't be placed in the block of the always statement which is to be mentioned later. Always describe the assign statement just under the module's block.

On the left-side of the assign statement, the operators explained later and the expression comsisting of the function statement can be written. Taking it as a description for the logical synthesis, it can be regarded that the description to create the "combinational circuit" is written on the right-side of the assign statement.

A.5 Operator

In the Verilog-HDL, the operators shown in Table A.2 are defined. (The operators not used in the logical synthesis are omitted partly.)
The operator's priority is shown in Table A.3. In the case that the operators of the same priority are shown in parallel, the operator appearing on the left-side of the expression is prioritized. (It is ithe positional relation not in the table, but in the expression of the Verilog-HDL description.)
However,explicit use of the paretheses is less erroneous than the implicit priority designation based on the table, and it is easy for other people to understand, too. Also, notice that the Table 1.3 on the page 22 in "Verilog-HDL Logical Circuit Design" [4] mistakenly mixes up the bit wise operation and reduction operation, giving a wrong description.

A.5.1 Arithmetic operation

Just like ordinary programming language, the operators of addition, substraction, division, multiplication and surplus have been prepared. The operation in the Verilog-HDL is always performed with no-signed integer. Among them, the operater with (△), addition, substraction, multiplication and surplus are logically synthesized to create a huge circuit, so be caruful about the operators. They are not used in this experiment.

A.5.2 Relational operation

This is the operation to check whether two values are equal (==) ot not (!=), or to check whether their large/small relation holds or not. The result becomes the value of one bit (1 or 0). Of course, the value is 1 when the relation holds.

Symbol	Meaning	Symbol	Meaning	Symbol	Meaning
+ /	**Addition** **Division (△)**	- %	**Addition** **Surplus**	*	**Multiplication (△)**
< > ==	**Smaller** **Larger** **Coincide**	<= >= !=	**Smaller than or eual** **Larger than or equal** **Not coincide**		
<<	**Left shift**	>>	**Right shift**		
!	**Logical negation**	&&	**Logical conjunction**	||	**Logical sum**
- ^	**Bitwise NOT** **Bitwise OR**	& ~^	**Bitwise AND** **Bitwise NOR**	|	**Bitwise OR**
& | ^	**Reduction AND** **Reduction OR** **Reduction XOR**	~& ~| ~^	**Reduction NAND** **Reduction NOR** **Reduction XNOR**		
?:	**Conditional Operation**	{ }	**Concatenation operation**		

Table A.2 List of operator

Priority	Operator
High	**Unary operation ! & ~& | ~| ^ ~^ + -**
	*** / %**
	Binary operation + -
	<< >>
	< <= > >=
	== !=
	Binary operation& ^ ~^
	Binary operation|
	&&
	||
Low	**?:**

Table A.3 Priority of operator

A.5.3 Logical operation and bitwise operation

Similar in form, the logical operation (! && ||) is the one for the logical value of 1 bit, while the bitwise operation (~ & | ^ ~^) is what the bit-by-bit operation is performed for the value of multi-bits.

A.5.4 Reduction operation

The reduction operation is what acts on the headof multi-bit signal line (or the item expressing the balue of multi-bit), and expresses the logicasl operation with all of the bits as an input, The result of operation is always 1 bit.
Example of reduction operation

```
wire [15:0]        bus;
wire [3:0]         a, b;
wire               c, d;

assign c = ~& bus;          (1)
assign d = ^(a + b);         (2)
```

In (1), the NAND (AND + NOT) for all the bits of the 15 bits signal line 'bus' is calculated and that is substituted to 'c'. In (2), after calculating the bitwise OR for the 4 bit signal lines of 'a' and 'b', the XOR for the values of 4 bits is calculated and substituted to 'd'.

A.5.5 Conditioned operator

The conditioned operator (?:) is a ternary operator, which takes the following 3 equations as an argument.

1. **Equation expressing the conditions.**
2. **Equation to be evaluated when the condition holds.**
3. **Equation to be evaluated when the condition doesn't hold.**

Its meaning is just like the description of arguments above. This is used to switch the value of a signal line according to conditions.
Example of conditioned operator

```
wire sel;                         // Selection signal
wire [3:0] input_A, input_B;      // Two data
wire [3:0] output;                // Ouput

// When the sel is 1, substitute (connect) the input_A to the output,
//When the sel is 0, substitute (connect) the input_B to the output,
assign output = sel ? input_A : input_B;
```

A.5.6 Concatenation operator

The concatenation operator is a notation that bundles plural signal lines to one signal line of multi-bits, and is used as follows.

Example of concatenation operator 1

```
wire [7:0]        opcode;
wire [3:0]        opra, oprb;
wire [15:0]       word;
wire [15:0]       bus;
wire [7:0]        addrh, addrl;

assign { opcode, opr_a, opr_b } = word;        (1)
assign bus = { addrh, addrl };                 (2)
```

The (1) is the example that the concatenation operator is used for the left-side of the assign statement, and it substitutes the 16 bits value, word, to the signal lines of 8 bits, opcode, that of 4 bits, opra and that of 4 bits, oprb, respectively. The (2) is the example that the operator is used for the right-side of the assign statement, and it combines two 8 bits signal lines, addrh and addl, and connect to the 16 bits signal line, bus. These examples use the concatenation operator in the assign statement, but this concatenation operation can be used in wherever ordinary multi-bits signal line is written.

Also, the repetition of the same patterns can be expressed in the following format. The concatenation operation with the repetition uses two pair of { }. Next to the first {, describe the repetition number, then describe the repetition pattern within the second { }.

Example of concatenation operation 2

```
wire [9:0]        pat0;
wire [15:0]       word16;
wire [7:0]        word8;

assign pat0 = { 5{ 2'b10} };                         (1)
assign word16 = { { 8{ word8[7]} }, word8 };         (2)
```

In the (1), the value of 10'b1010101010 is substituted to the pat0. The (2) is a little complicated and it copies the value of the 8th bit in the word[8] to the upper 8 bits of the word16. The lower 8 bits are the direct copy of the word8. This is called "sign extention", and used when extended to 16 bits with keeping the sign, in the case that the word8 is the 2's complement notation.

A.6 if statement

The format of the if statement is shown below.

if statement

```
if ( <Conditional expression> )        <Block 1>

or

if ( < Conditional expression > )      <Block 1>
else    <Block 2>

or

if ( < Conditional expression 1> )     <Block 1>
else if ( < Conditional expression 2> )    <Block 2>
else    <Block 3>
```

The 1 bit expression is described in the conditioned expression. In the block, a simple sentence or complex sentence surrounded by begin and end is described. However, because it is no problem even if the simple sentence is surrounded by the begin and end, the sentences should be always surrounded by the begin and end, in the consideration the readability and the case that a simple sentence may be altered to a complex sentence in the future.

Mind that the 'if' statement can be used only inside the function statement and always statement mentioned later.

A.7 case statementand casex statement

The format of the case statement is shown below.
case statement

```
case ( <Expression> )
<Value 1>: <Block 1>
<Value 2>: <Block 2>
.
.
.
default: <Default block>
endcase
```

In the case that the expression's value is 'value 1', The 'block' is executed. (That means the circuit to be executed is synthesized from the standoiunt of the logical synthesis.) This block is a simple sentence or comples sentence surrounded by begin and end, just like the case of if statement. The last default is described for the special case. If not corresponding to any cases above, this default block is executed. The line of the default can be omitted. However, in the case that the line of default: doesn't exist and other lines doesn't cover all of the cases, the result of synthesis becomes indefinite, so the line of default: should be incorporated in the case that many values may be used.

The casex statement belongs to the same type of syntax as the case statement. This can describe the 'X' indicating 'don't care value', in the description of a value.

Example of casex statement

```
casex ( in )
8’b1xxx_xxxx: penc = 3’d7;
8’b01xx_xxxx: penc = 3’d6;
8’b001x_xxxx: penc = 3’d5;
8’b0001_xxxx: penc = 3’d4;
8’b0000_1xxx: penc = 3’d3;
8’b0000_01xx: penc = 3’d2;
8’b0000_001x: penc = 3’d1;
8’b0000_0001: penc = 3’d0;
default: penc = 3’d0;
endcase
```

The priority encoder is described in this example. The input (in) is 8 bits. Among these, if the 7th bit is 1, 7 is output, regardless of other bits’ values , and if the 7th bit is 0 and the 6th bit is 1, 6 is output ,regardless of other bits’ values. In order to describe this circuit using the case statement, all the pattern of 28 = 256 should be described, and that is not efficient.

Mind that both thecase statement and the casex statement can be used only inside the fuctional statement and the alwaus statement.

A.8 function statement

A variety of expressions are described using the operators mentioned so far, but ther are many combinational circuits that can’t be described using the logical and arithmetic operations only. Such cases, the circuits are designed using the function statement.

Example of function statement 1

```
wire [1:0]          in;
wire [3:0]          out;

function [3:0]      decoder;
input [1:0]         f_in;
begin
case (f_in)
0: decoder = 4'b0001;
1: decoder = 4'b0010;
2: decoder = 4'b0100;
3: decoder = 4'b1000;
endcase
end
endfunction
assign out = decoder(in);
```

This example expresses, in the case that the signal line for the 2 bits input in the 4 bits output is regarded as a binary code, the circuit (decoder) which sets 1 only to the bit of the corresponding numerical (for example, 2'b10 corresponds to 2). For example, if the input is 2b19, the output is the pattern where only the second bit is 1, that is (4'b'0100).

The function statement first makes the range designation for the function's output, next describe the function name (decoder in this example). Subsequently the declaration of the input to the function is made. A function can take plural values. That case, the turn this statement appears corresdponds to the signal line calling it. Also, in the case that variables are used inside the function, they are defined by the reg declaration. The signal line defined outside of the function statement apparently seems usable inside the function. Actually this may cause errors, so when the signal line outside the function is to be used inside the function, describe the signal line as an argument of the function explicitly.

The assign statement and always statement can't be used inside the function statement. The '=' is used for the substitution. The other statements that can be used are the if statement, case statement and casex statement. As a value of function to be returned when the function statement is carried out, the value of variable with the same name as the function defined in the function statement is used. This variable is a special one and is not needed to be declared. (If declared, it causes an error.)

Inside a function, prosess is regarded to be carried out sequentially. See the example below.

Example of function statement 2

```
w wire [3:0] x;
wire [1:0] y;
wire [3:0] d;

function [3:0] f;
input [3:0] a;
input [1:0] b;
reg [3:0] c;
begin
c = a;
if ( b == 2'b01 ) begin
c = c + 1;
end
f = c << 1;
end
endfunction

assign d = f(x, y);
```

In this description on function, if thevalue of y is not 2'b01, the f returns x*2 and if the value of y is 2'b01, it returns (x+1)*2. Note that though the variable of f is not declared, it is automatically declared as the function name is f.

A.8.1 function statement in Verilog2001

In Verilog2001, the function can be defined by the following format.

Example of function statement 2

```
wire [1:0] in;
wire [3:0] out;

function [3:0] decoder(input [1:0] f_in);
begin
case (f_in)
0: decoder = 4’b0001;
1: decoder = 4’b0010;
2: decoder = 4’b0100;
3: decoder = 4’b1000;
endcase
end
endfunction

assign out = decoder(in);
```

Just like the module statement, the declaration of input is written in the parentheses next to the function name. This format is easier to understand.

A.9 always statement

The always statement is used for various purposes. Here, only the always to describe the synchronous sequential circuit is explained. In this experiment, use only the pattern appearing in the description

A.9.1 Description of D-Flip Flop

The example below is the description of D-Flip Flop.

D-Flip Flop

```
module dff(clk, d, q);
input clk, d;
output q;

reg q;

always @(posedge clk)
begin
q <= d;
end

endmodule
```

In the always statement, the event to trigger the execution of this always block is described after the

‘@’. In this example, the ‘posedgeclk’ corresponds to the event, and it means “when the value of the signal line, ‘clk’, rises up from 0 to 1”. After that, the main body’s block is described. Equal to the block of if statement and case statement, this is a simple sentence or plural sentence surrounded by begin and end. Itshould be always surrounded by begin and end, to prevent an error.

Though the if statement, case statement and casex statement can be used inside the always block, the assign statement and function statement (the definition of function) can’t be described, Namely, the definition of assign and function always exists, and this is not what is executed event by event. The instance description mentioned later can’t be performed in the always block, too. Also, inside the always block, the assignmet is written in the notation ‘<=’. This format of assignment is called non-blocking assignment. In the example above, every time the clk changes from 0 to 1, the alwaus block is executed and the value of the input d is assigned to the output q. Also, the signal line in the left-side of the non-blocking assignment should be reg-declared. In the example above, if the wire declaration and reg declaration are omitted, it is regarded that the wire declaration is made implicitly, so it turns out that the reg declaration is made explicitly.

The next example is the description on the series connection of two D-FlipFlops.

Series connection of D-FlipFlop

```
module dff2(clk, in, out);
input clk, in;
output out;

reg q1;
reg out;

always @(posedge clk)
begin
q1 <= in;
out <= q1;
end

endmodule
```

The circuit diagram synthesized by this description is shown in Figure A.1.

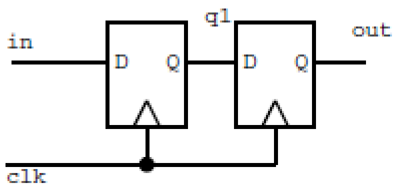


Figure A.1 Series connection of D-FlipFlop

The substitution statement in programming languages is called blocking substitution (to distinguish from non-blocking. Namly, as far as the process of one assignment statement is not finished, the operation is not moved to the next statement. (It is blocked.) On the other hand, the non-blocking assignment moves theoperation to the next sentence not waiting for the completion of the assignment

processing, so it can be regarded as all the statements are being executed at the same time, in some sense. In the example above, it is not that after the 'q1 <= in' process is done, the 'out <= q1' process is conducted subsequently, but it is that the processes of 'q1 <= in' and 'out <= q1' are executed at the same time. Namely, the value of 'q1' one clock before, that is, the value of 'in' 2clock before, is in the 'out'. If the substitution is made successively, it becomes equivalent of the 'out <= in' and this description turns out to be the same as that of the D-FlipFlop above. The non-locking assignment doesn't give any meaning to the turn appearing in its description, so even if the turn of the assign statement is

```
out <= q1;
q1 <= in;,
```

the entirely same circuit is synthesized.

It is syntactically possible to have plural always statements in a module. But even if there are plural D-FlipFlops, only one always statement should be created in the description of one module, as the description that the logical synthesis is possible.

A.9.2 Description of D-FlipFlop with non-synchronous rest

The following example is the description of D-FlipFlop with non-synchronous rest.
D-FlipFlop

```
module dff(clk, rst, d, q);
input clk, rst, d;
output q;

reg q;

always @(posedge clk or negedge rst)
begin
if ( !rst )
q <= 0;
else
q <= d;
end

endmodule
```

The meaning of this description is that when the block's booting timing is the clk's rising up or the rst's falling down (transition from 1 to 0), 0 is assigned to q if the rst is 0, otherwise d is assigned q, just like the D-FlipFlop without reset.

Hereafter, upon describing the sequential circuit, the following format is to be used.

```
always @(posedge clk or negedge rst)
begin
        if ( !rst ) begin
                initialization assignment on the resetting
        end
        else begin
                Ordinary operation
        end
end
```

Note in particular that if the 'else' next to 'if (!rst) begin ~ end' is mistakenly omitted, the synthesis can't be processed.

The description of the circuit where two units of this D-FlipFlop with non-synchronous reset is connected in series.

Series connection of D-FlipFlop with non-synchronous reset

```
module dff2(clk, rst, in, out);
input clk, rst, in;
output out;

reg q1;
reg out;

always @(posedge clk or negedge rst)
begin
if ( !rst ) begin
q1 <= 0;
out <= 0;
end
else begin
q1 <= in;
out <= q1;
end
end

endmodule
```

This is different from the description without the reset, in the points that the 'negedge rst' is added next to '@' of always and the part of if (!rst) is added.

A.9.3 Description of sequential circuit including combinational circuit

Of course, it is also describable the sequential circuit with complicated operation in addition to the simple assignment of values. The following example is the description of the 4 bits binary counter with non-synchronous reset.

4 bits binary counter with non-synchronous reset.

```
module count4(clk, rst, q);
input           clk, rst;
output [3:0]    q;

reg [3:0] q;

always @(posedge clk or negedge rst)
begin
if ( !rst )
begin
q <= 4’b0000;
end
else
begin
q <= q + 1;
end
end
endmodule
```

The next example is the description that the example above is improved to the decimal counter. (This example intentionally omits the part of ‘begin ~ end’ as much as possible. Confirm about the case that such a description is needed.

Decimal counter with non-synchronous reset

```
module count10(clk, rst, q);
input           clk, rst;
output [3:0]    q;

reg [3:0]       q;

// Calculate the decimal counter’s next value.
function [3:0]  next;
input [3:0];

begin
next = in + 1;
if ( next == 10 )
next = 0;
end
endfunction

always @(posedge clk or negedge rst)
if ( !rst )
q <= 4’b0000;
else
q <= next(q);
endmodule
```

A.10 Instance description

Using the syntax elements explained so far, basic combinational circuit and sequential circuit are describable as a module. However, it isn’t eficcient to describe a large scale circuit by one module, and it tends to be errornous. The Verilog-HDL provides the framework to build a larger module utilizing the existing module as a part. This is the same concept as the circuits designed on circuit diagram editor is symbolized so as to be used for other circuit diagrams. What a module is used for parts as such is called ‘instantiation’ in the Verilog-HDL. The next example describes a 4 bits adder using the full adder as a part. In this example, for the convenience of explanation, the upper- and lower case letters are used to distinguish between the the signal line of the lower module (full adder) and that of the upper module (adder4). But it is not favorable to distinguish signal lines using such a case sensitivity, in general.

Example of instantiation description

```
module fulladder(A, B, CIN, S, COUT);
input A, B, CIN;
output S, COUT;

assign S = A ^ B ^ CIN;
assign COUT = A & B | A & CIN | B & CIN;
endmodule

module adder4(a, b, cin, s, cout);
input [3:0] a, b;
input cin;
output [3:0] s;
output cout;

wire tmp1, tmp2, tmp3;

fulladder fa1(a[0], b[0], cin, s[0], tmp1);
fulladder fa2(a[1], b[1], tmp1, s[1], tmp2);
fulladder fa3(a[2], b[2], tmp2, s[2], tmp3);
fulladder fa4(a[3], b[3], tmp3, s[3], cout);

endmodule
```

The circuit diagram corresponding to this description is shown in Figure A.2.

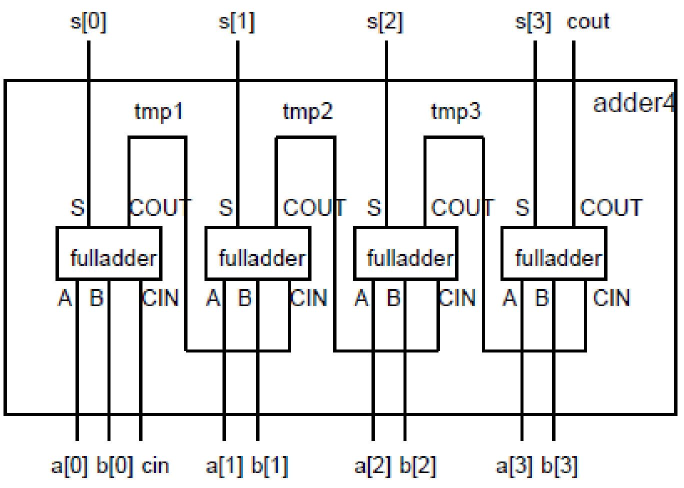


Figure A.2 4 bits adder

The description on the instantiation is written in the format as follows.

<Module name> <Instance name>(Signal line name 1, Signal line name 2, ...)

Here, the ‘module name’ is the original name of the lower module to be instantiated. (It is ‘fulladder’ in the example of Fig. A.2.) The ‘instance name’ the name to distinguish individual instances. This

example creates the instances of 4 fulladders, so they are named as fa1, fa2, fa3 and fa4, respectively, for the distinction.

Actually, the declaration of wire tmp1, tmp2 and tmp3 in this description is not necessary. Such the 1 bit wire type signal line is declared implicitly. However, in the cae of the implicit declaration, even if a mistake in writing like
'fulladder fa2(a[1], b[1], temp1, s[1], tmp2);' (The 'tmp1' is mistakenly written 'temp1'.)
occurs, the compiler doesn't report the error on undefined signal line name, and this may delay the bug finding. Though such misdescription should be automatically checked by a compiler, describers need to check it carefully by themselves.

In the example of instantiated module, the signal lines are assigned to module's inout/output ports in order, while in the following example, they are assigned to the ports explicitly deignated. Though the latter case tends to make the description longer and irksome, it can make the mistakes in the assignment to the ports decrease.

Example of instantiation description designating port's name

```
fulladder   fa1(.A(a[0]),   .B(b[0]),   .S(s[0]),   .CIN(cin),   .COUT(tmp1));
```

In this example, the connection is deacribed in the following format.
.<Name of port in lower module>(<Name of signal line connected to port>)

Anyway, the instatiation description for lower modules is in the same level as the description of circuit using circuit diagram, so this doesn't give a good readability which is a feature of the hardware describing language. It is necessary to devise such a way as preparing for a simple circuit diagram (block diagram) beforehand then describing that in Verilog-HDL, or checking carefully whether the assignment of port names to signal lines is correct or not (though the hardware describing language should be a bit wiser). Also, in the case of the lower module with simple description, Direct description of the lower module's content into the upper module may be easier to understand, rather than the instantiation of the lower module.

Another important point in the instatiation description is that the signal line connected to the lower module's output port should be wire-type one. It is natural, considering that the connection to the lower module's output port is a kind of assign statement and the upper module's signal line corresponds to the left-side of assign statement. Because the signal line connected to an input port is the so-called right-side of assign statement, both the wire type and reg type are acceptable this case [*1].

A.11 Points to be noted on Verilog-HDL description

Finally, it is summarized the points tobe noted on the Verilog-HDL description once again.

***1 Actually, it is not only limited for single wire or reg, but also acceptable for an expression. Because this may cause mistakes, do not try it until get accustomed to the way to use.**

- Designate the bit width for a numeric value.

- There are wire-type and reg-type in the signal line.

- The left-side of assign statement is wire-type, whichever will be fine for the right-side.

- The left-side of non-blocking assignment (<=) is reg type.

- Variables inside functuion statement should be reg-type, but the '=' is used for the substitution.

- The assign statement and instance description are not written in the always statement.

- The if statement, case statement can be written only in the function statement.

- One 'always@(posedge clk or negedge rst)' for one module.

- The signal line connectable to the output port in the instance description is the wire-type one.

- Even if describing undeclared signal line due to typography, it is implicitly regarded as the wire-type signal line of 1 bit, so make a visual check carefully.

- Be careful about the correspondence in begin ~ end and that in if ~ else.

- Rules only for the case of logical synthesis using the Quartus :
The project name should be the same as the name of the uppermost module. Also, one module should be described in one file and the file name should be 'module name'.v.

- Guideline to be observed, though it is not the rule.

- One module is to be described in one file.

- The file name should be 'module name'.v.

- In addition to these issues, when encountering any odd things, ask to teachers and TAs

Appendix B Specifications for FPGA board

The appendix B describes the specifications for the FPGA board of MU500 (MU500-RX, MU500-RK andMU500-7SEG).

B.1 Outline

The MU500-RX is the FPGA board [*1] loaded with the Cyclon IV that is the FPGA manufactured by ALTERA (Intel). Through the upper and lower connectors, it is connected to the user interface board MU500-RK loaded with key switches and the MU500-7SEG loaded with a lot of 7SEG-LED.

Figure B.1 shows the names for each part of MU500-RX, MU500-RK and MU500-7SEG .

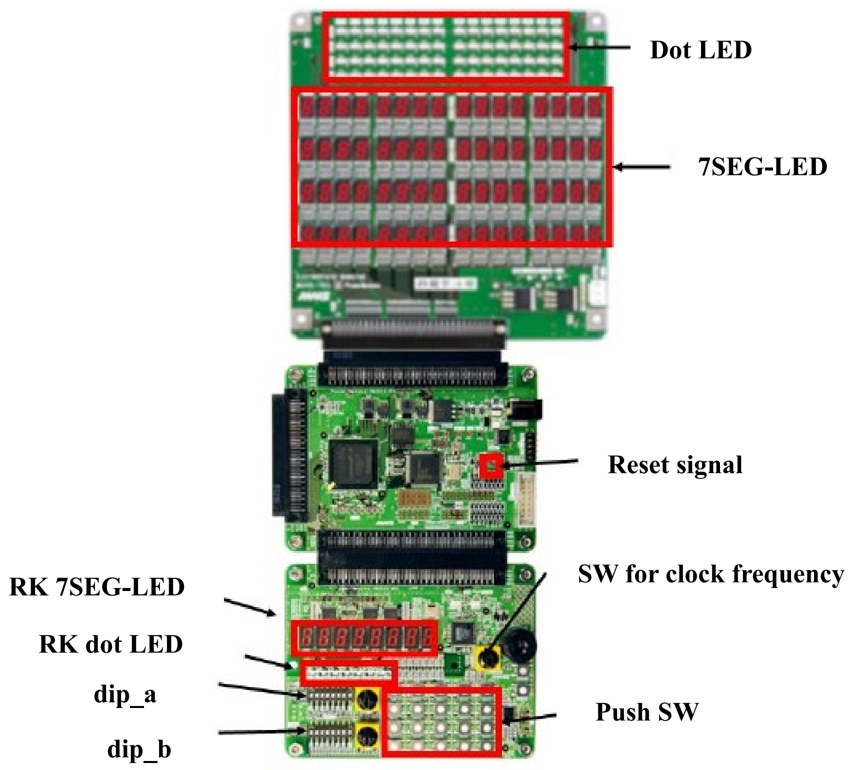


Figure B.1 Names for each part of FPGA board

[*1] Besides that, the FPGA board is also loaded with my computer RX210 manufactured by Renesus, but it is not used in this experiment.

The switch used on the MU500-RX board at the center is the reset switch (impressed with RESET on a white coat) only at the right-center (the reset signal in Figure B.1). Also, the LED indicating energization state is mounted near an electric plug.

The MU500-RK board at the lower part has the following interfaces.

- **20 pieces of ten key switch**
- **3 rotary switches**
- **8 pieces of 7-segmanted LED**
- **8 pieces of dot LED**

Among them, the rotary switch at right-center is used for the purpose of setting clock frequency. The correspondence between switch's value and frequency is shown in Table B.1.

Table B.1 Correspondence between CK_DIV and clock frequency

CK_DIV	0	1	2	3	4	5	6	7
Frequency	40MHz	20MHz	10MHz	5MHz	1.25MHz	312.5kHz	78.1kHz	9.8kHz

CK_DIV	8	9	A	B	C	D	E	F
Frequency	9.8kHz	4.9kHz	2.44kHz	1.22kHz	610Hz	305Hz	1.0Hz	*

The case of CK_DIV (* in the table) is a special one. When pressing the push button of CLKSW at the right edge of the board, one piece of pulses is applied to the clock signal.

Other switches and LEDs are connected to the FPGA as they are, and can be freely utilized by preparing appropriate circuits in the FPGA side. Because this experiment uses the same pin assignment in all of the exercises, use the following identifier's names in the definition of top level modules.

Table B.2 Input/output declaration for MU500-RX and MU500-RK

Verilog-HDL description	Explanation
input sys_clock	Clock for whole system. Frequency is fixed to 20MHz.
input reset	Reset signal directly connecting to the RESET button. Note that it works in negative logic. (Turn to 0 by pressing the switch.)
input clock	Clock signal whose frequency can be changed by the SW for clock frequency.
input psw_a0, —, psw_a4	5 switches in the 1st line (horizontal direction) of ten key. Note that it works in negative logic. (Turn to 0 by pressing the switch.)
input psw_b0, —, psw_b4	5 switches in the 2nd line (horizontal direction) of ten key. Note that it works in negative logic. (Turn to 0 by pressing the switch.)
input psw_c0, —, psw_c4	5 switches in the 3rd line (horizontal direction) of ten key. Note that it works in negative logic. (Turn to 0 by pressing the switch.)
input psw_d0, —, psw_d4	5 switches in the 4th line (horizontal direction) of ten key. Note that it works in negative logic. (Turn to 0 by pressing the switch.)
input [3:0]]hex_a	Output of the upper one of 2 rotary switches in a queue. It expresses the values of 0 – F(15) with signal line of 4 bits.
input [3:0]]hex_b	Output of the lower one of 2 rotary switches in a queue. It expresses the values of 0 – F(15) with signal line of 4 bits.
input [7:0] dip_a	Output of the upper one of 2 DIP switches in a queue. The values of 8 switches are expressed by the signal line of 8 bits. It turns to 1 when the switch is at the upper side.
input [7:0] dip_b	Output of the lower one of 2 DIP switches in a queue. The values of 8 switches are expressed by the signal line of 8 bits. It turns to 1 when the switch is at the upper side.
output [7:0] seg_x output [3:0] sel_x	Control signal for 4 pieces (A, B, C and D) at left-half of 8 pieces of 7SEG-LED on RK board. Details are described later.
output [7:0] seg_y output [3:0] sel_y	Control signal for 4 pieces (A, B, C and D) at right-half of 8 pieces of 7SEG-LED on RK board. Details are described later.
output [7:0] led_out	Signal line connecting to 8 dot LEDs in a queue.

The correspondence of the push SWs to the name of signal lines is shown in Figure B.2. Mind that these push SWs work in the negative logic. They are usually 0, and turn to 1 only when they are pressed.

psw_a0	psw_a1	psw_a2	psw_a3	psw_a4
psw_b0	psw_b1	psw_b2	psw_b3	psw_b4
psw_c0	psw_c1	psw_c2	psw_c3	psw_c4
psw_d0	psw_d1	psw_d2	psw_d3	psw_d4

Table B.2 Lay-out of push SWs

The MU500-RX and MU500-7SEG have the 7SEG-LED and dot LED, respectively. Hereafter, in order to distinguish them, what are monted on the MU-500RK are called 'RK 7SEG-LED' and 'dot LED', respectively. What are mounted on the MU500-7SEG are called '7SEG-LED' and 'dot LED', as they are.

The 7SEG-LED actually has 7 segments and one dot (at the position of decimal point), so 8 bits are needed to control one LED. In the case of controlling 8 pieces of th 7SEG-LED simultaneously, it is necessary to use the signal lines of 64 (= 8 x 8) bits. However, the pin number of the FPGA is in the order of several hundred ~ one thousand, and it is not efficient that 64 pins are given to the 8 pieces of 7SEG-LED. Then, in the MU500-RK, the group of 4 pieces of a 7SEG-LED shares 4 segment-signals (seg_x and seg_y), and the select sigmal (sel_x and sel_y) indicates which of the 7SEG-LEDs is driven by the present segment-signal. Namely, only one of 4 pieces of the 7SEG-LED cab be driven simultaneously, so the circuit to control output signal with time sharing is needed to display the 7SEG-LED.

The bit assignment for each segment is shown in Figure B.3.

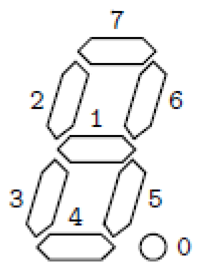
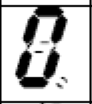
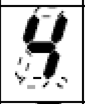
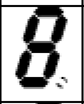
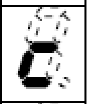
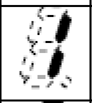
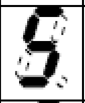
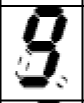
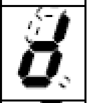
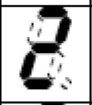
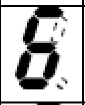
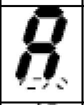
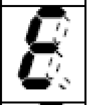
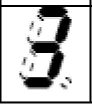
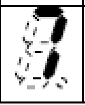
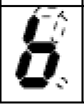
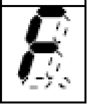


Figure B.3 Bit assignment for 7-segmented LED

Table B.3 shows an example of the pattern expressing numerical.

Table B.3 Example of numeral pattern for 7-segmented LED

a[7:0]		a[7:0]		a[7:0]		a[7:0]	
1111_1100		0110_0110		1111_1110		0001_1010	
0110_0000		1011_0110		1111_0110		0111_1010	
1101_1010		1011_1110		1110_1110		1001_1110	
1111_0010		1110_0000		0011_1110		1000_1110	

The lay-out of the RK 7SEG-LED is shown in Figure B.4.

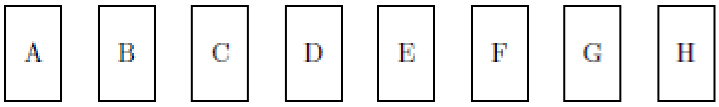


Figure B.4 Lay-out of RK 7SEG-LED

The 8 RK dot LEDs just nelow the RK 7SEG-LED is in a similar lay-out, and it is bundled together to the the signal line of 8 bits, led_out. As a top level interface. Ths correspondence of each bit with LED is shown in Table B.4.

Table B.4 Lay-out of RK dot LED

A	B	C	D	E	F	G	H
led_out[0]	led_out[1]	led_out[2]	led_out[3]	led_out[4]	led_out[5]	led_out[6]	led_out[7]

Next, the board for display, MU500-7SEG is outlined. The MU500-7SEG consists of 64 pieces of 7SEG-LED and 64 pueces of dot LED. Seeing from the FPGA side, all of them are regarded as an output. The identifier's name used in the Verilog-HDL description is shownin below.

Table B.5 Output declaration for MU500-7SEG

Verilog-HDL description	Explanation
input [7:0] seg_a	Segemnt signal of A group (left-half of the 1st line)
input [7:0] seg_b	Segemnt signal of B group (right-half of the 1st line)
input [7:0] seg_c	Segemnt signal of C group (left-half of the 2nd line)
input [7:0] seg_d	Segemnt signal of D group (right-half of the 2nd line)
input [7:0] seg_e	Segemnt signal of E group (left-half of the 3rd line)
input [7:0] seg_f	Segemnt signal of F group (right-half of the 3rd line)
input [7:0] seg_g	Segemnt signal of G group (left-half of the 4th line)
input [7:0] seg_h	Segemnt signal of H group (right-half of the 4th line)
input [8:0] sel	Selection signal

The framework is almost the same as that of the 7SEG-LED of MU500-RK. This case, the 8 pieces of the 7SEG-LEDs are brought together to one group. Further, the similar grouping was made for the 64 pieces of dot LEDs, so each group shares 9 types of segment-signal in total, that is, 8 pieces of the 7SEG-LED and the one for the LED of 8 bits. Therefore the selection signal becomes 9 bits. Because this is also displayed b time sharing, the control circuit to drive it is needed.

B.2 Display driver for MU500-RK and MU500-7SEG

As mentioned before, The MU500-RK and 7SEG-LED of MU500-7SEG share the segment signal in time division, so it is necessary to control the process using the sequence circuit. In this experiment, the existing one designed beforehand is used for this display driver circuit. The external specifications are shown in the following.

Display driver circuit for MU500-RK and MU500-7SEG

```
module led_driver(input          sys_clock,   // System clock
                  input          reset,       // Reset
                  input [63:0]   seg7_a,      // Segment signal of MU500-7SEG's A group
                  input [63:0]   seg7_b,      // Segment signal of MU500-7SEG's B group
                  input [63:0]   seg7_c,      // Segment signal of MU500-7SEG's C group
                  input [63:0]   seg7_d,      // Segment signal of MU500-7SEG's D group
                   input [63:0]  seg7_e,      // Segment signal of MU500-7SEG's E group
                   input [63:0]  seg7_f,      // Segment signal of MU500-7SEG's F group
                   input [63:0]  seg7_g,      // Segment signal of MU500-7SEG's G group
                   input [63:0]  seg7_h,      // Segment signal of MU500-7SEG's H group
                   input [63:0]  seg7_dot64,  // Output signal of MU500-7SEG's dot LED
                   input [7:0]   rk_a,        // A segement signal of MU500-RK
                   input [7:0]   rk_b,        // B segement signal of MU500-RK
                   input [7:0]   rk_c,        // C segement signal of MU500-RK
                   input [7:0]   rk_d,        // D segement signal of MU500-RK
                   input [7:0]   rk_e,        // E segement signal of MU500-RK
                   input [7:0]   rk_f,        // F segement signal of MU500-RK
                   input [7:0]   rk_g,        // G segement signal of MU500-RK
                   input [7:0]   rk_h,        // H segement signal of MU500-RK
                   output [7:0]  seg_a,       // Board output (seg_a) of MU500-7SEG
                   output [7:0]  seg_b,       // Board output (seg_b) of MU500-7SEG
                   output [7:0]  seg_c,       // Board output (seg_c) of MU500-7SEG
                   output [7:0]  seg_d,       // Board output (seg_d) of MU500-7SEG
                   output [7:0]  seg_e,       // Board output (seg_e) of MU500-7SEG
                   output [7:0]  seg_f,       // Board output (seg_f) of MU500-7SEG
                   output [7:0]  seg_g,       // Board output (seg_g) of MU500-7SEG
                   output [7:0]  seg_h,       // Board output (seg_h) of MU500-7SEG
                   output [8:0]  sel,         // Board output (sel) of MU500-7SEG
                   output [7:0]  seg_x,       // Board output (seg_x) of MU500-RK
                   output [3:0]  sel_x        // Board output (sel_x) of MU500-RK
                   output [7:0]  seg_y,       // Board output (seg_y) of MU500-RK
                   output [3:0]  sel_y);      // Board output (sel_y) of MU500-RK
...
endmodule
```

Those of the seg7_a, seg7_b, seg7_c, seg7_d, seg7_e, seg7_f, seg7_g and seg7_h are the signal lines expressing the content to display on the 8 pieces of 7SEG-LED, respectively. As 8 bits are needed for one piece of 7SEG-LED, one group (8 pieces of 7SEG-LED) uses 64 bits in total. The 'seg7_dot64' is an output signal of 64 dot LEDs and it is indexed to 0, 1, … from the upper-left to the right direction. The outputs to the seg_a ~ seg_h and the sel are made by switching these inputs appropriately in time sharing. Also, those of the rk_a, rk_b, rk_c, rk_d, rk_e, rk_f, rk_g and rk_h are the signal lines expressing the content to display on the 8 pieces of 7SEG-LED. The outputs to the seg_x, sel_x, seg_y and sel_y are done by switching the signal lines appropriately in time sharing.

Appendix C Specifications for educational processor KAPPA3-RV32I Ver. 0.1

C.1 Outline

This document describes the specifications for educational processor KAPPA3-RV32I (Kyushu Advanced Program for Processor Architecture Ver. 3 -RV32I). This processor aims to provide the minimum function needed for education and to have a structure which is easily understood by students. As a basic architecture, it is adopted the open architecture RISC-V(RV32I), that is a load/store type architecture with a general register, in order to facilitate design works and to promote the understanding on the progress toward the implementation by pipeline. There are several variations in RISC-V, according to the type of word length and instruction set. Here, it is used the RV32I supporting only the 32 bits integer operation instruction. Just like other general RISC (Reduced Instruction Set Computer) processors, all of arithmetic logic operations are conducted between registers or between a register and an immediate value. Main specifications are described below.

Word length : - 32 bits constitute one word.

- Byte addressing --- The memory address is assigned to each unit of 1 byte (8 bits). That is, one word consists of 4 bytes.

- Little endian --- The format where the lower bytes in the 4 bytes are assigned to lower address. Namely, writing the value of 32 bits, 32'h12345678 (notation of Verilog-HDL) into the four bytes starting from the address 0 (that is, writing into the address 0 to 3), the value at the address 0 becomes 8'h78 in lower 8 bits, the value at the address 1 becomes 8'h56 in next 8 bits … and the value at the last address 3 becomes 8'h12.

- All the access should be aligned. The access to a word of 4 bytes should be made for the address of multiples of 4. The access to a half-word of 2 bytes should be made for the address of multiples of 2 without fail.

Internal memory : The FPGA tip has an internal memory of 64 Kbytes. As the description designed in advance is given, there is no need for students to design it.

Timer : There is a counter of 64 bits operating independently from the processor. Upon the timer's count value coinciding with a designated value, an interrupt request is generated.

Interrupt : The processor has the function of simple interrupt. According to the interrupt request, the normal execution of the program is suspended and another program is executed.

The KAPPA3-RV32I complies with the specifications for RISC-V's RV32I, but the processor is too complicated to use in the experiment for students, as it is. Therefore, The KAPPA3-LIGHT has been

prepared, that simplifies the KAPPA3-RV32I by removing the functions related to the interruption. In this experiment, the design and operation check of this KAPPA3-LIGHT are to be conducted at first, and subsequently the practice on software development using the KAPPA3-RV32I is to be conducted. Hereafter, the specifications for the KAPPA3-RV32I are described. They are the same as those for the KAPPA3-LIGHT except the description on the interrupt and CSR (to be mentioned later).

C.2 Instruction format

All the instructions are the 32 bits (one word) -fixed length and have the following six type formats. Here, regardless of the instruction format, the lowest seven bits, which are the field called 'opcode', indicate the type of instructions.

Table C.1 Type of instruction format

31 27 26 25	24 20	19 15	14 12	11 7	6 0	Instruction
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

R-type : Mainly used for register-register operation instruction. R_d is called destination register. It designates the register storing the operation result or the loaded value. The detail function is designated by the funct3 field of remaining 3 bits and the funct7 field of 7bits.

I-type : Used for register-immediate value operation, register indirect jump (JALR) and load instruction. R_d is called destination register. Though it usually designates the register storing operation results, it is used to designate the register storing return values (PC's present values) in the case of a jump instruction. R_{s1} is called destination register. It designates the register used for the operation. In the case of the jump instruction and load instruction, it used as the base register expressing the address. An immediate value of upper 12 bits is used as a value for the operation in the case of the operation instruction, and used as an offset of address in the case of the load instruction and jump instruction.

S-type : Used in store instruction. R_{s1} register is used as a base register for address calculation. R_{s2} expresses the value to be stored. Just like the load instruction, the immediate value is expressed by 12 bits that extend over two fields. This is devised so that the designated fields for the registers of R_{s1}, R_{s2} and R_d should be made the same, in all instructions.

B-type : Used in the branch instruction. Though similar to the S-type, the encoding of immediate value is different. This is devised to use the immediate value shifted by 1 bit, because the address of

branch destination is even number so the lowest bit (the 0th bit) becomes unnecessary. However, except the 11th bit and 12th bit of the immediate value, it is encoded in the same way as the S-type.

U-type : Used in the upper load instructions (LUI and AUIPC). This type uses the 20 bits except R_d field and opcode, as an immediate value. Mind that this expresses the upper 20 bits from the 31st bit to 12th bit.

J-type : Used in the jump instruction. Just like the U-type, this type also uses the 20 bits except R_d field and opcode, as an immediate value, but it is different in encoding. This is devised so that the bits from 10th bit to 1st bit of the immediate value should be the same as the I-type. As the address of the jump destination is even number, the lowest bit always become zero, so it is not designated.

C.3 Addressing mode

The addressing mode is the method to designate the position in a memory (memory address). Roughly saying, only one type of addressing mode is supported in KAPPA3-RV32I. This is given in the form of 'the value of designated register (base register)' + 'offset'. In this regard, there are 4 types for the type of base register and the form of offset, as follows.

- **I-type : Use R_{s1} as a base register. The 12 bits immediate value is sign extended and added to the base register's value.**
- **S-type : Like the I-type, R_{s1} is used as a base register. The 12 bits immediate value is sign extended and added to the base register's value. But the field of immediate value is different.**
- **B-type : Though not clearly indicated in the instruction, the PC (program counter) is used as a base register. Further, after sign extended, the 12 bits immediate value is doubled and added to the value of the PC.**
- **J-type : This type also uses the PC as a base register. The J-type's immediate value is 20 bits. Just like the S-type, it is doubled and added to the PC after sign extended.**

In this way, the load/store instruction uses a general register as a base register, and the branch/jump instruction uses the PC as a base register. However, only the JALR instruction is exceptional, and the general register designated in the R_{s1} field is used as a base register. Only this instruction can make the process jump into the address irrelevant to the present PC, wherever a program is arranged. The reason why other branch/jump instructions use the address relevant to the PC is to make the revise in designation of branch destination's address under instruction, unnecessary. The code, that is no need to rewrite the content according to the position arrangement like this, is called PIC (position independent code). By using the PIC code, the procedure of linker/loader used in program development is largely simplified.

C.4 Instruction set

C.4.1 Load and jump instruction for immediate value

Table C.2 Instruction set (1)

31 27 26 25	24 20	19 15	14 12	11 7	6 0	Instruction
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

LUI (Load Upper Immediate) : U-type

Load an immediate value at the upper 20 bits of register. The lower 12 bits are always zero.

AUIPC (Add Upper Immediate to PC) : U-type

Add an immediate value to the PC. The immediate value is what the upper 20 bits are designated. The lower 12 bits are regarded to be added zero.

JAL (Jump And Link) : J-type

Enter the present PC's value (its own address + 4) into R_d, and jump to the designated address.

JALR (Jump And Link Register) : I-type

Enter the present PC's value (its own address + 4) into R_d, and jump to the address of R_{ds1} + *imm*, where the *imm* is an immediate value designated in the immediate field. The *nimm* is treated as an integer of signed 12 bits.

BEQ (Branch EQual) : B-type

Jump to the address that the immediate value is added to the present PC's value (its own address + 4) in the case of $R_{s1} = R_{s2}$. The immediate value is treated as an integer of signed 13 bits. Mind that the encoding of the immediate value in this format is complicated. For the instructions that only the branch condition is different, there are BNE (Branch Not Equal: branch for $R_{s1} \neq R_{s2}$), BLT (Branch Less Than: branch for $R_{s1} < R_{s2}$), BGE (Branch Greater Than or Equal: branch for $R_{s1} \geq R_{s2}$), BLTU (Branch Less Than Unsigned: branch for $R_{s1} < R_{s2}$, regarded as no-signed integer) and BGEU (Branch Greater Than or Equal Unsigned: branch for $R_{s1} \geq R_{s2}$, regarded as no-signed integer).

C.4.2 Load/Store instruction

LB (Load Byte) : I-type

Read out the value of 1 byte from a memory. The address to be read out is designated by R_{s1} + *imm*, where the imm is a signed 12 bits integer designated in the field of immediate value. The read-out value is <u>sign extended</u> and entered into R_d.

Table C.3 Instruction set (2)

31 27 26 25	24 20	19 15	14 12	11 7	6 0	Instruction
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

LH (Load Half word) : I-type

Read out a value of 2 bytes (half word) from memory. The address to be read is designated by R_{s1} + *imm*, where the imm is a signed 12 bits integer designated in the field of immediate value. The read-out value is <u>sign extended</u> and entered into R_d. Address should be an even number.

LW (Load Word) : I-type

Read out a value of 4 bytes (word) from memory. The address to be read is designated by R_{s1} + *imm*, where the imm is a signed 12 bits integer designated in the field of immediate value. The read-out value is entered into R_d. Address should be multiples of 4.

LBU (Load Byte Unsigned) : I-type

Read out a value of 1 byte from the memory. The address to be read is designated by R_{s1} + *imm*, where the imm is a signed 12 bits integer designated in the field of immediate value. The read-out value is entered into R_d, as it is. Zero is entered into the upper 24 bits.

LHU (Load Half word Unsigned) : I-type

Read out a value of 2 bytes (half word) from the memory. The address to be read is designated by R_{s1} + *imm*, where the imm is a signed 12 bits integer designated in the field of immediate value. The read-out value is entered into R_d, as it is. Zero is entered into the upper 16 bits.

SB (Store Byte) : S-type

Write a value of 1 byte into the memory. The address to be written is designated by R_{s1} + *imm*, where the imm is a signed 12 bits integer designated in the field of immediate value. The R_{s2} is used for the value to be written.

SH (Store Half word) : S-type

Write a value of 2 bytes (half word) into the memory. The address to be written is designated by R_{s1} + *imm*, where the imm is a signed 12 bits integer designated in the field of immediate value. The R_{s2} is used for the value to be written. The address should be an even number.

SW (Store Word) : S-type

Write a value of 4 bytes unto the memory. The address to be written is designated by R_{s1} + *imm*, where the imm is a signed 12 bits integer designated in the field of immediate value. The R_{s2} is used for the value to be written. The address should be multiples of 4.

There exist 5 types of load instructions of LB, LH, LW, LBU and LHU. Among them, the instructions with 'B' for the second character (LB and LBU) make the access in the unit of a byte (Byte : 8 bits). The instructions with 'H' for the second character (LH and LHU) make the access in the unit of a half word (Half Word : 16 bits). The instructions with 'W' for the second character make the access in the unit of a word (Word : 32 bits). The instructions with 'U' for the third character (LBU and LHU) treat the read-out value as a no-signed number. The other instructions with two characters (LB, LH and LW) treat the read-out pair as a signed number. What the number is signed or no-signed affects whether the values of 8/16 bits are sign extended or not. Suppose that the value read out in 8 bits is 8'b1111 1111 [*1]. Regarding it as a no-signed number, it becomes 255 in the decimal code. On the other hand, regarding it as a signed number, it becomes -1 in the decimal code. Extended it to 32 bits, they become 32'b0000 0000 0000 1111 1111 and 32'b1111 1111 1111 1111 1111, respectively.

As for the store instruction, just like the load instruction, there exist 3 types of SB, SH and SW, according to the unit to be accessed. However, in the case of writing, there is no distinction between signed and no-signed because the bit extension is not made.

As the instruction word length is 32 bits in KAPPA3-RV32I, it is efficient if the access to the memory can also be made in the unit of 32 bits. But in the case that the memory access in the unit of 8 bits and 16 bits, the following consideration will be needed. First, suppose that data of one byte (8 bits) is to be loaded (read-out) from the address of 32'h8765 4321. Though the address of the memory is assigned for each byte, the address range accessed is 4 bytes from 32'h8765 4320 to 32'h8765 4323 because the data of 32 bits (= 4 bytes) are actually treated as one block. (See Figure C.1.)

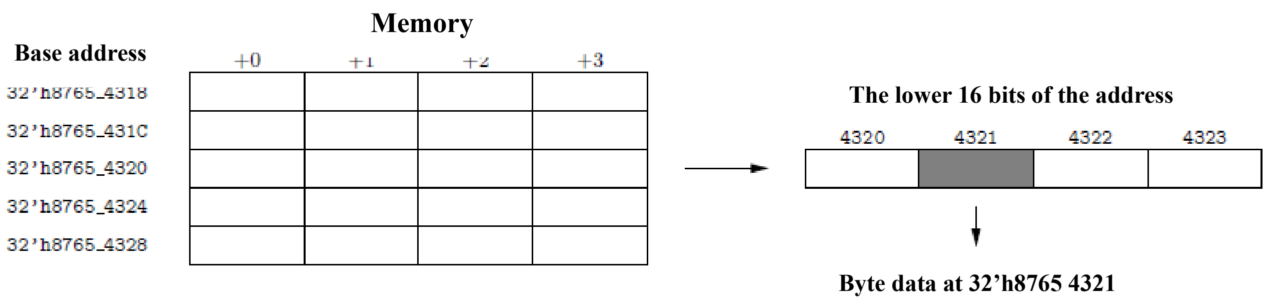


Figure C.1 Example of loading data in the unit of byte

Among the data, only the second byte is needed, so in the LB instruction it is necessary to read out the data of 32 bits once, and then to make the process of segmenting the data of 8 bits. The similar process should be made for the case of loading the data in the unit of 16 bits.

Some ingenuity is needed for the store instruction. Consider storing data to the address of 32'h8765 4321 in the unit of a byte, just like the example above. This case, the content in the addresses of 32'h8756 4320 and 32'h8756 4322 shouldn't be rewritten. As a simple method, it is considered the procedure that the 4 bytes' data from the address of 32'h8765 4320 to 32'h8765 4323 is read-out once

----

*1 Notice that '_' in numeric expression is ignored in Verilog-HDL. It is used for a 4 bits separator here.

to temporary storage place, only the second byte in the 4 bytes is rewritten and then written back to the original place. However, it is not efficient because one reading-out and one writing-in are needed to execute one store instruction. Therefore, devising the memory side, prepare for a bit mask to designate a byte for writing-in. Specifically, add the 4 bits input-signal-line, wrbits, to a memory. When writing into the memory, the data should be written into only the byte with this wrbits of 1. In the previous example, as only the second byte is written, the wrbits turns out to be 4'b0010. In the ordinary case of writing to all of 4 bytes (32 bits), the wrbits should turn out to be 4'b1111.

C4.3 Operation instruction for immediate value

Table C.4 Instruction set (3)

31 27 26 25	24 20	19 15	14 12	11 7	6 0	Instruction
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

ADDI (ADD Immediate) : I-type

Calculate '$R_{s1} + imm$' and store it into R_d. The *imm* is what the 12 bits signed integer designated in the field of immediate value is extended to 32 bits.

SLTI (Set Less Than Immediate) : I-type

If $R_{s1} < imm$, enter 1 into R_d. Otherwise, enter 0. The *imm* is what the 12 bits signed integer designated in the immediate value field is extended to 32 bits.

SLTIU (Set Less Than Immediate Unsigned) : I-type

If $R_{s1} < imm$, enter 1 into R_d. Otherwise, enter 0. The imm is what the 12 bits signed integer designated in the immediate value field is extended to 32 bits. Here, the large-small comparison is made by regarding the data as no-signed integer.

XORI (XOR Immediate) : I-type

Calculate '$R_{s1} \oplus imm$' and store it to *Rd*. The *imm* is what the 12 bits signed integer designated in the immediate value field is extended to 32 bits. The operation of $\oplus$ calculates the exclusive-OR for every bit.

ORI (OR Immediate) : I-type

Calculate '$R_{s1} \vee imm$', and enter it to R_d. The *imm* is what the 12 bits signed integer designated in the immediate value field is extended to 32 bits. The operation of $\vee$ calculates the OR for every bit.

ANDI (AND Immediate) : I-type

Calculate '$R_{s1} \wedge imm$' and enter it into R_d. The *imm* is what the 12 bits signed integer

designated in the immediate value field is extended to 32 bits. The operation of ∧ calculates the AND for every bit.

SLLI (Shift Left Logical Immediate) : I-type

Store the result that the value of R_{s1} is logically left-shifted (to the direction of the lowest bit), into R_d. Because the shift amount 31 bits at maximum (If shifting 32 bits, nothing remains.), the lower 5 bits (shamt) in the immediate value field are used. In the case of SLLI, the upper 7 bits are always zero. In the logical shift, zeros are entered into the bits becoming empty by shifting.

SRLI(Shift Right Logical Immediate) : I-type

Store the result that the value of R_{s1} is logically right-shifted (to the direction of the lowest bit), into R_d. Because the shift amount 31 bits at maximum (If shifting 32 bits, nothing remains.), the lower 5 bits (shamt) in the immediate value field are used. In the case of SRLI, the upper 7 bits are always zero. In the logical shift, zeros are entered into the bits becoming empty by shifting.

SRAI(Shift Right Arithmetic Immediate) : I-type

Store the result that the value of R_{s1} is arithmetically right-shifted (to the direction of the lowest bit), into R_d. Because the shift amount 31 bits at maximum (If shifting 32 bits, nothing remains.), the lower 5 bits (shamt) in the immediate value field are used. In the case of SRAI, the upper 7 bits are always 0100000. In the arithmetic shift, the value of the uppermost bit is entered into the bit becoming empty by shifting. Namely, regarded as signed integer, the sign is not changed by shifting.

C4.4 Instruction for register operation

Table C.5 Instruction set (4)

31 27 26 25	24 20	19 15	14 12	11 7	6 0	Instruction
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

ADD (ADD) : R-type

Store the result of '$R_{s1} + R_{s2}$' into *Rd*.

SUB(SUB) : R-type

Store the result of '$R_{s1} - R_{s2}$' into *Rd*.

SLL(Shift Left Logical) : R-Type

The value of R_{s1} is logically shifted to left by R_{s2}. In the logical shift, zero is entered to the bit which becomes empty by shifting.

SLT (Set Less Than) : R-type

When $R_{s1} < R_{s2}$, enter 1 into R_d. Otherwise, enter 0. The data is regarded as signed integer in large and small comparison.

SLTU (Set Less Than Unsigned) : R-Type

When $R_{s1} < R_{s2}$, enter 1 into R_d. Otherwise, enter 0. The data is regarded as no-signed integer in large and small comparison.

XOR (XOR) : R-type

Store the result of $R_{s1} \oplus R_{s2}$ into R_d. The operation of $\oplus$ calculates the exclusive-OR for every bit.

SRL (Shift Right Logical) : R-Type

The value of R_{s1} is logically right-shifted by R_{s2}. In the logical shift, zero is entered to the bit which becomes empty by shifting.

SRA (Shift Right Arithmetic) : R-Type

The value of R_{s1} is arithmetically right-shifted by R_{s2}. In the arithmetic shift, the value of the uppermost bit is entered into the bit which becomes empty by shifting. Namely, when regarded as signed integer, the sign is not changed by shifting.

OR(OR) : R-type

Store the value of the result for '$R_{s1} \vee R_{s2}$' into R_d. The operation of $\vee$ calculates the OR for every bit.

AND(AND) : R-type

Store the value of the result for '$R_{s1} \wedge R_{s2}$' into R_d. The operation of $\wedge$ calculates the AND for every bit.

C.5 Privileged instruction and interrupt processing

The privileged instruction and interrupt processing in KAPPA3-RV32I are described here. These functions are not implemented in KAPPA3-LIGHT. The following four privilege levels are supposed in RISC-V.

- **Machine mode**
- **User mode**
- **Supervisor mode**
- **Hypervisor mode**

Among them, only the machine mode is implemented in KAPPA3-RV32I. All of the privileged instructions are executable in the machine mode. Though vulnerable in security, the implementation is simple because the management and shift of privilege level are not included.

The privilege instructions in RISC-V are shown in Table C.6. The instructions used in other modes than the machine mode are omitted.

Table C.6 Instruction set (3)

31 27 26 25	24 20	19 15	14 12	11 7	6 0	**Instruction**
0011000	00010	00000	000	00000	1110011	MRET
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

MRET (Machine mode RETurn): R-Type

Return from the interrupt processing in the machine mode. Specific operations are described later.

CSRRW (CSR Read and Write): I-Type

CSR reading & writing.

CSRRS (CSR Read and Set): I-Type

CSR reading & bit set.

CSRRC (CSR Read and Clear): I-Type

CSR reading & bit clear.

CSRRWI (CSR Read and Write Immediate): I-Type

CSR reading & immediate value writing

CSRRSI (CSR Read and Set Immediate): I-Type

CSR reading & immediate value bit set

CSRRCI (CSR Read and Clear Immediate): I-Type

CSR reading & immediate value bit clear

The instruction beginning with CSR is the access instruction of CSR (control status register). Strictly speaking, it is not the privileged instruction, but included in this category because it is closely related to the processing of privilege level. Actually instruction's opcodes for MRET and CSR group are the same value of 110011, and they are designed as the same instruction group.

Table C.7 CSR of KAPPA3-RV32I

Address	Mnemonic	Meaning	Remarks
0x300	MSTATUS	Overall status	Only the MSTATUS[3] and MSTATUS[7] have a meaning.
0x304	MIE	Interrupt permission bit mask	Only the MIE[7] has a meaning.
0x305	MTVEC	Interrupt table address	
0x340	MSCRATCH	Temporary register for interrupt handler	
0x341	MEPC	Return address for interrupt	
0x342	MCAUSE	Cause of interrupt	Always 32'h8000 0007
0x344	MIP	Interruption waiting state	Only the MIP[7] has ameaning.

Interrupt/exception in RISC-V are as follows.
- **Access fault exception**
- **Breakpoint exception**
- **Environment call exception**
- **Illegal instruction exception**
- **Unaligned address exception**
- **Software interrupt**
- **Timer interrupt**
- **External interrupt**

In KAPPA3-RV32I, only the timer interrupt in the machine mode is treated, for simplification.

C.5.1 CSR and CSR controlling instructions

The CSR (Control Status Register), which consists of many 32 bits registers, is mainly used for the control on the privileged instruction and interrupt. Though it is a register file designated by the 12 bits address in the instruction set, it is not to all of the 4096 (= 2^{12}) addresses that meaningful registers are assigned. It is also included what is useless in the implementation of the privileged mode and floating point arithmetic. The CSR to be implemented in KAPPA3-RV32I is shown in Table C.7. The mnemonic is a name given for the register in a specific address. Hereafter, CSR registers are referred using this mnemonic. For example, the CSR[0x300] is 'nMSTATUS'. Also, the MSTATUS[3] expresses the 3rd bit of MSTATUS register (Verilog-HDL notation).

Though the MSTATUS has a 32 bits length, this time only the two bits of MSTATUS.MPIE = MSTATUS[7] and MSTATUS.MIE = MSTATUS[3] are used. Actually only the two bits are implemented as a register and the remaining always returns 0 for read-out. The MSTATUS.MIE is a

flag that is set to 1 when permitting the interrupt on the machine mode. The MSTATUS.MPIE is a register to keep an original value when clearing the MSTATUS.MIE in the interrupt handler on the machine mode.

The MIE is a bit mask indicating the permission/non-permission (e of enable) for each interrupt cause. This time, the MIE.MTIE = MIE[7] is implemented because only the timer of the machine mode is implemented. Other bits always return 0.

The MIP is abit mask indicating whether each interrupt is pending or not (p of pending). Because only the timer on the machine mode is implemented just like the MIE, only the 'MIP.MTIP = MIP[7]' is implemented. Other bits always return 0. The writing into MIP.MTIP is not permitted.

MTVEC keeps the address to jump when the interrupt occurs. To be exact, there are vector modes of the jumping to the address indicated in the MTVEC for the case of MTVEC[1:0] = 2'b00, and of the jumping to the address of MCAUSE ×4 from the base address obtained by clearing the lower 2 bits of the MTVEC for the case of MTVEC[1:0] = 2'b01. In KAPPA#-RV32I, the vector mode is not implemented.

The MSCRATCH is used for keeping the value of one general register at the start of the interrupt handler's process. At the same time, the head (bottom) address of the stack area the interrupt handler uses is retained beforehand. This enables other general registers to be saved to the stack area for the interrupt handler.

The MEPC retains the value of the PC when the interrupt occurs. This is used as the address to be returned, in executing the return instruction MRET from the interrupt handler.

The MCAUSE expresses the interrupt cause. The timer interrupt of the machine mode is 32'h8000 0007.

Because the CSR is mapped in the memory space different from ordinary memories, a distinctive access instruction has been prepared. All the instructions are the I-type and have the fields for R_{s1}, R_d and *Imm* (12 bits). Among them, the *Imm* is used as the address of CSR. The R_{s1} usually expresses the source register, but in the instructions related to the immediate value, csrrwi, csrrsi and csrrci, it is used as the 5 buts immediate value (zimm). The value is used after 0 is extended to the upper 27 bits.

The operation of each instruction is shown below.

- CSRRW	**Read out the value of CSR to R_d, and write the value of R_{s1} into CSR.**
- CSRRS	**Read out the value of CSR to R_d, and write the bitwise OR with the value of R_{s1} into CSR.**
- CSRRC	**Read out the value of CSR to R_d, and write the bitwise AND with the inverse of R_{s1}'s value into CSR.**
- CSRRWI	**Read out the value of CSR to R_d, and write the immediate value into CSR.**
- CSRRSI	**Read out the value of CSR to R_d, and write the bitwise OR with the immediate value into CSR.**
- CSRRCI	**Read out the value of CSR to R_d, and write the bitwise AND with the inverse of immediate value into CSR.**

Because the immediate value has only 5 bits in the instruction related to the immediate value, it is unable to set a value in the upper 27 bits of CSR.

C5.2 Interrupt processing

In RISC-V, the interrupt processing is done in the case of MSTATUS.MIE = 1 ∧ MIE.MTIE = 1 ∧ MIP.MTIP = 1. Specifically, the following processes are conducted.

- MEPC <= PC
- Set the cause in MCAUSE. It is fixed to the timer interrupt in the machine mode.
- MIP.MTIP <= 0
- MSTATUS.MPIE <= MSTATUS.MIE
- MSTATUS.MIE <= 0
- PC <= MTVEC

By setting the MTVEC's value to the PC, the process moves to the routine of the interrupt processing in the next execution of instructions. At the end of the interrupt processing routine, the process returns to the original program by the MRET instruction. The followings are the concrete procedure in that case.

- PC <= MEPC
- MSTATUS.MIE <= MSTATUS.MPIE

In order to implement the return MRET instruction from the interrupt, judge in the DE phase whether it is the MRET instruction or not. In the case of the MRET instruction, the WB phase transits to the IF phase after restoring the PC's value, as described above.

C5.3 Timer

The timer is the hardware for counting time independent of executing instructions, and it is accessed from the RISC-V through the address mapped in ordinary memory space. Here, two 64 bits registers of mtime and mtimecmp are prepared. The mtime is the register for reading-out only, that is initialized to 0 on resetting and then added by 1 for every one cycle (system clock). The mtimecmp is the readable/writable register, and when its value coincides with mtime's one the timer interrupt occurs. That is, MIP.MTP becomes 1.

C.6 Configuration

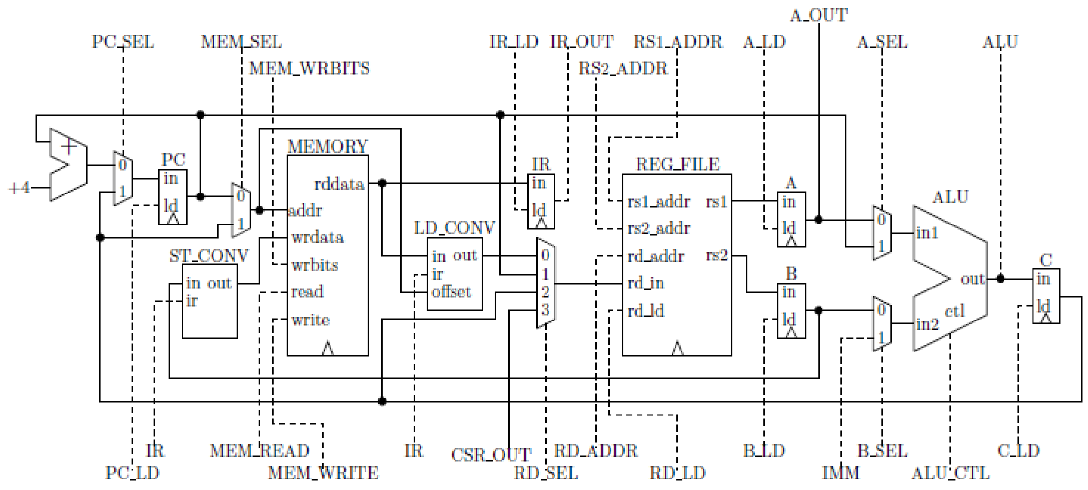


Figure C.2 Configuration of KAPPA3-RV32I

The logical configuration of KAPPA3-RV32I is shown in Figure C.2. The KAPPA3-RV32I is roughly divided to the following components. Note that the broken lines extending upward or downward indicate the input/output to the controller. The clock signal and reset signal for the flip-flop (register) are omitted. The signal lines observing and setting values of each register for debugging are also omitted, the detail of which is mentioned later. The CSR is also omitted. 'A_out' and several control inputs are employed for the input to the CSR. The output from the CSR is used dor the input to the RD_SEL as the CSR_OUT.

PC Program counter. This retains an address of the instruction to be executed next. It uses a general 32 bits register. The control relevant to the PC is as follows.
- PC_LD : Set 1 when writing a value to the PC.
PC_SEL The selecter (multi-plexer) control to select the input to the PC is as follows.
- 0 : 'PC + 4' is used.
- 1 : The value of the C register is used.
Memory This stores the memory, instruction program and data. Designate the memory address to access to the ADDR. Designatethe value to be written into the DTA. The read-out value is output to the OUT. The control relavant to the Memory is as follows.
- MEM_READ : Set 1 when reading a value of the memory is read out.
- MEM_WRITE : Set 1 when writing a value into the memory.
- MEM_WRBITS : The bit mask designating an object to be written into the memory
MEM_SEL The selecter control to select a memory's address is as follows.
- 0 : The value of the PC is used.
- 1 : The value of the C register is used.
IR The register retaining the instruction being executed currently. A general 32 bits register is employed. The read-out value is output to IR_OUT. The control relevant to the IR is as follows.

- IR_LD : Set 1 when writing a value into the IR.

REG_FILE : Register file It has 32 registers of 32 bits, R0 ~ R31, in order to temporarily store the data. The group of same type general registers like this is called the register file. It is equipped with two reading ports (RS1 and RS2) and one writing port (DATA).

- **RS1_ADDR : Designate the register number to be read out by the RS1.**
- **RS2_ADDR : Designate the register number to be read out by the RS2.**
- **RD_ADDR : Designate the register number to be written into.**
- **RD_LD : Set to 1 when writing into the register.**

The R0 register is a special one that the read-out result always becomes zero and nothing is done in writing (not to be an error).

RD_SEL The selector control to select the input to be written into the register file is as follows.

- **0 : Use the memory's output.**
- **1 : Use the PC register's value.**
- **2 : Use the C register's value.**
- **3 * Use the CSR's output (only in KAPPA3-RV32I).**

A, B Two registers of 32 bits to retain the values read out from the register file. The A register stores the value of the register designated in R_{s1} field, and the B register stores the one designated by R_{s2}. The control related to the A and B register is as follows.

- **A_LD : Set to 1 when writing a value into the A register.**
- **B_LD : Set to 1 when writing a value into the B register.**

The A register's value (A_OUT) is used for the input to the CSR, too.

ALU This is the core part conducting operations. The ALU itself is a combinational circuit without a memory. The ALU has two inputs of 32 bits and an output of 32 bits. The control related to the ALU is as follows.

- **ALU_CTL : Designate the operation conducted in the ALU. Refer to Appendix C.5 for the detail.**

ASEL This is the selector to select an input of the ALU's input 1 (in1). The control is as follows.

- **0 : Use the A register's value.**
- **1 : Use the PC register's value.**

BSEL This is the selector to select an input of the ALU's input 2 (in2). The control is as follows.

- **0 : Use the B register's value.**
- **1 : Use the immediate value (IMM).**

C This is a 32bits register to retain the ALU's operation result. Setting the CLD to 1, the ALU's value is written into.

CSR CSR (Control Status Register). It is omitted in Figure C.2. Refer to C.5.1 for the detail. It has the CSR_ADDR to designate an object to be accessed, and has the the input/output of the CSR_OP to designate the input value (CSR_IN) and the operation (Write-Set-Clear).

- **The CSR_ADDR makes the connection to the IMM.**
- **The CSR_IN makes the A_OUT or RS1_ADDR connected, according to circumstances.**
- **The CSR_OP is genaerated by a controller.**
- **The CSR_OUT makes the connection to the input of the RD_SEL selector.**

C.7 Phase

The KAPPA3-RV32I executes one instruction by the operation of plural clocks, just like many processors. Here, the operation carried out by one clock is called "phease". For the simplification of the KAPPA3-RV32I, the same phase structure is used for all of instructions.

IF (Instruction Fetch) : Instruction reading phase.

The following process is performed.

1. **Substitute the value of the memory in the address pointed by the PC into the IR.**
2. **Add the PC's value by 4 (in the unit of 32 bits).**

DE (Decode) : Instruction analysis phase.

The following process is performed.

1. **Enter the value of the register shown in the R_{s1} field in IR into the A register.**
2. **Enter the value of the register shown in the R_{s2} field in IR into the B register.**

EX (EXcute) : Execution phase.

The following process is performed.

1. **In the case of the operation instruction, perform the operation and enter the result into the C Register.**
2. **In the case of the load, store and jump instructions, perform the address calculation and enter the result into the C register.**
3. **In the case of the store instruction, enter the value to be written (R_d) into the C register.**

WB (Write Back) : Memory/write back phase.

The following process is performed.

1. **In the case of the operation instruction, write the C register's value into the register designated by R_d.**
2. **In the case of the load instruction, read out the value from the memory and write it into the register designated by R_d.**
3. **In the case of the store instruction, write the C register's value into the memory.**
4. **In the case of the jump instruction, write the C register's value into the PC register.**
5. **In the case of the branch instruction, check the branch condition. If the condition holds, enter the C register's value into the PC register.**

IR (InterRupt) : Interrupt phase.

This is the phase executed when an interrupt request is made. The following procedure is performed.

1. **MEPC <= PC Retain the PC address to return.**
2. **Set the cause in the MCASE. In this procedure, fix to the timer interrupt of the machine mode.**
3. **MIP.MTIP <= 0 Clear the timer interrupt request.**
4. **MSTATUS.MPIE <= MSTATUS Retain the interruption-permitted-state.**
5. **MSTATUS.MIE <= 0 Prohibit a new timer interrupt.**
6. **PC <= MTVEC Set the address of the interrupt handler in the PC.**

In each phase, the IF and DE phases perform the same procedure regardless of a type of instruction. Other phases perform different procedures according o instructions.

Though the operation of ALU is usually performed only in the EX phase, in the BEQ (Conditioned branch instruction) the address calculation for the branch destination is performed in the EX phase and the judgement of branch condition is performed in the WB phase.

The instructions of CSR series are mainly processed in the CSR. The ALU is notnused.

Usually the procedure of ‘IF -> DE -> EX -> WB -> IF …’ is repeated. However, in the case that the interrupt request is made, the IR phase is carried out before moving from the WB to the IF (only for the KAPPA3-RV32I). Because the PC’s value is changed in the IR phase, the process moves to the interrupt handler.

Table C.8 Phase table

Instruction	IF	DE	EX	WB
ADD group			C $\leftarrow$ A + B	reg[R_d] $\leftarrow$ C
ADDI group			C $\leftarrow$ A + I_imm	reg[R_d] $\leftarrow$ C
Load group			C $\leftarrow$ A + I_imm	reg[R_d] $\leftarrow$ mem[C]
Store group	IR $\leftarrow$ mem[PC]	A $\leftarrow$ reg[R_{s1}]	C $\leftarrow$ A + S_imm	mem[C] $\leftarrow$ B
LUI	PC $\leftarrow$ PC + 4	B $\leftarrow$ reg[R_{s2}]	C $\leftarrow$ U_imm	reg[R_d] $\leftarrow$ C
AUIPC			C $\leftarrow$ PC + U_imm	reg[R_d] $\leftarrow$ C
JAL			C $\leftarrow$ PC + J_imm	reg[R_d] $\leftarrow$ PC PC $\leftarrow$ C
JALR			C $\leftarrow$ A + I_imm	reg[R_d] $\leftarrow$ PC PC $\leftarrow$ C
BEQ			C $\leftarrow$ PC + B_imm	if (A == B) PC $\leftarrow$ C
MRET				PC $\leftarrow$ MEPC MSTATUS.MIE $\leftarrow$ MSTATUS.MPIE
CSRRW				reg[R_d] $\leftarrow$ CSR[I_imm] CSR[I_imm] $\leftarrow$ A
CSRRWI				reg[R_d] $\leftarrow$ CSR[I_imm] CSR[I_imm] $\leftarrow$ zimm

C.8 Input/display module for KAPPA3-LIGHT

The specifications for controlling the operation of the KAPPA3-LIGHT and observing its internal state are described. Refer to the Appendix B regarding to the description of the FPGA board itself.

C.8.1 Mode

There are two modes - input mode and operation mode – in the KAPPA3-LIGHT. The input mode is the mode to write a value into a memory and register on the KAPPA3-LIGHT, and it can be entered by turning on the DIP A-0 of the DIP switch. The operation mode is the mode to execute a program, and it can stop the program at phase-by-phase and instruction-by-instruction. The operation mode can be entered by turning off the DIP A-0. In the DIP switch, The upside of the DIP switch is on and the downside is off. More specifically, only the operation of right-side one row of the push SW differs by the input mode and the operation mode.

C.8.2 Switch and LED

clock **This is the rotary switch to adjust the clock frequency. If the clock is too fast, a misoperation occurs at the time of key input. Therefore, it shoud be set at an appropriate speed so that the flickering of LED synchronized to the clock cycle can be visually observed. On the contrary, if the clock is too slow, no response occurs by pushing a key. Usually use the clock with the dial of C ~ E.**

Rest **Reset switch. The reset signal in circuits is directly connected. This initializes all the registers to 0 except the memory.**

Push switch **This consists of 16 numeric keys from 0 to F and 4 keys of 'clear', '+', '-' and '='. The numeric keys are used for entering a value to the input buffer. The right-side 4 keys works differently according to the modes (Table C.9).**

Table C.9 Fuction of keys

Input mode		Operation mode	
'clear'	clear	-	unused
'+'	Add to an address by 4	SP	Execution/stop
'-'	Subtract from an address by 4	SI	Stop by every phase
'='	Write	SS	Stop by every instruction

When the clear ley is pressed, the value of the input buffer is cleared to 0. The register memory of the KAPPA3-LIGHT is not affected. The '+' and '-' are used to add or subtract the address (the value of MAR) for the access to a memory. Because one word is 32 bits (4 bytes) in the KAPPA3, the address increases or decreases in the unit of 4. When the '=' key is pressed, the value of the input buffer is written into the target register memory.

DIP_A-0 **TThis his is used for switching the input mode and operation mode. When this is 'off', the operation mode is selected, and when this is 'on', the the input mode is selected.**

HEX_A **This is used to observe the value of the memory and the internal register, and to designate the target element when inputting a value (Table C.10).**

HEX_B **This is used only to select a general register (Table C.10). However, this switch has a meaning only when a general register is designated in the HEX_A.**

Table C.10 Meaning of Rotary SW

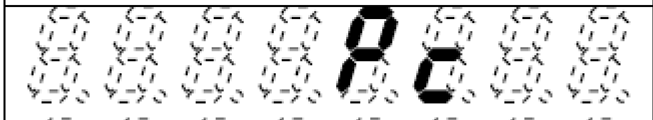
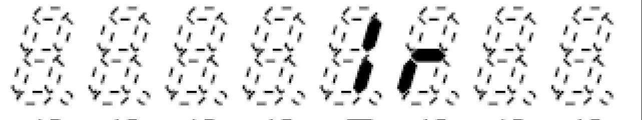
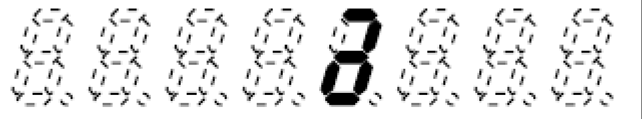
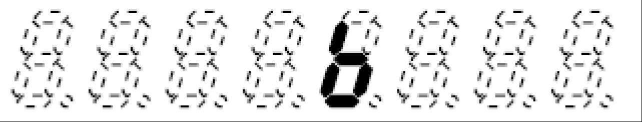
HEX_A	HEX_B	
0	—	PC
1	—	IR
2	—	A
3	—	B
4	—	C
5	—	
6	offset	reg[offset]]
7	offset	reg[16 + offset]
8	—	MAR
9	—	memory

RK-A ~ RK-H **Display the input buffer's value.**

7SEG-A ~ 7SEG-H **Display the content of the register memory. The left-side 4 groups indicate the object currently displayed. The right-side 4 groups indicate the actual values. Because only 8 objects can be displayed at one time, the displayed content differs according to the objects designated in HEX_A and HEX_B.**

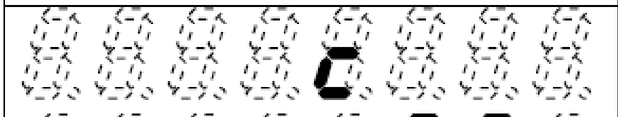
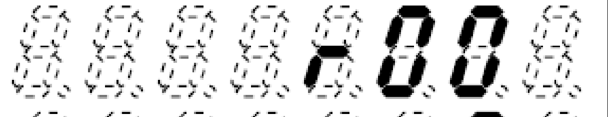
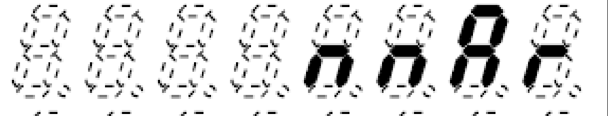
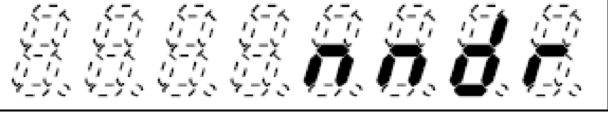
When the values of HEX_A are 0, 1, 2 and 3, the content is displayed as shown in Table C.11.

Table C.11 Page 1

Content displayed in left-side 7SEG-LED	Meaning	Content displayed in right-side 7SEG-LED
	PC	**PC's value**
	IR	**IR's value**
	AREG	**A register's value**
	BREG	**B register's value**

When the values of HEX_A are 6, 7, 8 and 9 and the value of HEX_B is 0, the content is displayed as shown in Table C.12.

Table C.12 Page 2

Content displayed in left-side 7SEG-LED	Meaning	Content displayed in right-side 7SEG-LED
	CREG	**C register's value**
	R00	**Register file's value (reg[0] in this example)**
	MAR	**MAR's value**
	MDR	**mem[MAR]'s value**

The R00 on the second line is actually displayed differently, according to the values of HEX_A and HEX_B. In the input mode, the 7SEG-LED for the current writing target is flickering slowly. Pressing the ten key, a value is entered into the 7SEG-LED of the MU500-RK side. Therefore, pressing the '=' key, a value is entered into the target register memory. When writing a value into a memory, set a memory address to the MAR at first, and then taking the writing target as a memory, enter the value. Because the keys of '+' and '-' increase/decrease the value of the MAR in the unit of 4, it is efficient when values are written into successive addresses.

C.9 pecifications for each part

C.9.1 General 32 bits register

Template for general 32 bits register

```
module reg32(input                clock,     // Clock
input               reset,     // Reset

input [31:0]        in,        // Data to be written
input               ld,        // Writing control signal

output reg [31:0]              out,       // Output

input               dbg_mode;             // Debugging mode
input [31:0]        dbg_in,               // Data to be written in debugging mode
Input               dbg_ld);              // Writing control signal in debugging mode
...
endmodule
```

Configuration **General 32 bits register used in PC, IR, A, B and C.**

Operation **Perform the process following the priority below.**

- If the reset is 0, set the internal value 32'b0.

- If the dbg_mode is 1 and the dbg_ld is 1, write the value of dbg_in.

- If the dbg_mode is 0 and the ld is 1, write the value of in.

This is described in the public/reg32.v.

C9.2 Register file

Temlate of register file

```
module regfile(input            clock,          // Clock signal (rising edge)
input           reset,          // Reset signal (reset with 0)

input [4:0]             rs1_addr,       // Register number of RS1
input [4:0]             rs2_addr,       // Register number of RS2
input [4:0]             rd_addr,        // Register number of RD

input [31:0]    rd_in,          // Data to be written into RD
input           rd_ld,          // Write-control-signal of RD

output [31:0]   rs1_out,        // Output of RS1
output [31:0]   rs2_out,        // Output of RS2

input           dbg_mode;       // Debugging mode
input [31:0]    dbg_in,         // Data written in debugging mode
input [4:0]             dbg_addr,       // Register number in debugging mode
input           dbg_ld,         // Write-control-signal in debugging mode
output [31:0]   dbg_out);       // Output in debugging mode
...
endmodule
```

Configuration This consists of 32 general registers of 32 bits. Here, ***R*0** is a dummy register that always returns 0 in reading and the value is unchanged in writing.

Operation The process is performed following the priority below.

- Output the value of the register designated by the dbg_addr to the dbg_out. As the dbg_out is not a r egister, use the assign statement.
- Output the value of the register designated by the rs1_addr to the rs1_out. As the rs1_out is not a register, use the assign statement.
- Output the value of the register designated by the rs2_addr to the rs2_out. As the rs2_out is not a register, use the assign statement.
- If the reset is 0, set the values of all registers zero.
- If the dbg_mode is 1 and the dbg_ld is 1, write the value of the dbg_in into the register designated by the dbg_addr.
- If the dbg_mode is 0 and the rd_ld is 1, write the value of the rd_in into the regiater designated by the rd_addr.

This is described in the public/regfile.v.

C.9.3 STCONV

Template of STCONV

```
module stconv(input [31:0]    in,      // Input
input [31:0]    ir,  // IR's value
output [31:0] out);            // Output
...
```

Configuration This is a combinational circuit to convert the 32 bits darta to be written into a memory. Used in the store instruction (sb, sh and sw).

Operation

- sb instruction : Copy the value of the lower 8 bits in the 'in' to an appropriate position. For example, in the case of writing into the address of 32'h0001, it is necessary to left-shift the in[7:0] by 8 bits, as {16'b0, in[7:0], 8'b0}. However, because the target to be written is actually designated by the mem_wrbits, the part not to be written is not needed to be 0. Then, in the case of the sb instruction, it should be used {in[7:0], in[7:0], in[7:0], in[7:0]} (or {4{in[7:]}}), regardless of the address to be written.
- sh instruction : Copy the value of the lower 16 bits in the 'in' to an appropriate position. Consider in the same way as tha case of the sb instruction.
- sw instruction : Enter the 'in' to the 'out', as it is.

C.9.4 LDCONV

Template of LDCONV

```
module ldconv(input [31:0]    in,             // Input
input [31:0]    ir,                    // IR's value
input [1:0]     offset,      // Lower 2 bits of memory address
output [31:0]   out);        // Output
...
```

Configuration **This is a combinational circuit to convert the 32 bits data read out from a memory. Used in the load instruction)lb, lbu, lh, lhu and lw).**

Operation **- lb instruction : As the value of 4 bytes including the relavant address is stored in the 'in', the corresponding byte data is taken out. <u>Further, it is sign-extended.</u>**

- lbu instruction : As the value of 4 bytes including the relavant address is stored in the 'in', the corresponding byte data is taken out. Here, enter 0 to the upper bits.

- lh instruction : As the value of 4 bytes including the relavant address is stored in the 'in', the corresponding data of 16 bits (a half word) is taken out. <u>Further, it is sign-extended.</u>

- lhu instruction : As the value of 4 bytes including the relavant address is stored in the 'in', the corresponding data of 16 bits (a half word) is taken out. Here, enter 0 to the upper bits.

- lw instruction : Enter the 'in' to the 'out', as it is.

C.9.5 ALU

Template of ALU

```
module alu(input [31:0]    in1,        // Input1
input [31:0]         in2,       // Input2
input [ 3:0]         ctl,       // Function-control- signal
output [31:0]        out);      // Output
...
endmodule
```

Configuration Implement as a pure combinational circuit.

Operation - Operation of ALU : This is operated as shown in Table C.13, following the value of the ctl.

Table C.13 Operation of ALU

ctl	Operation	Remarks
0000	out ← in2	For LUI
0010	out ← in1 == in2	For equivalent comparison (BEQ)
0011	out ← in1 ! = in2	For non-equivalent comparison (BNE)
0100	out ← in1 < in2	For less than comparison (BLT and SLT)
0101	out ← in1 >= in2	For greater than comparison (BGE)
0110	out ← in1 < in2	For unsigned less than comparison (BLTU and SLTU)
0111	out ← in1 >= in2	For unsigned greater than comparison (BGEU)
1000	out ← in1 + in2	For ADD and branch destination address calculation
1001	out ← in1 − in2	For SUB
1010	out ← in1 $\oplus$ in2	For XOR
1011	out ← in1 $\vee$ in2	For OR
1100	out ← in1 $\wedge$ in2	For AND
1101	out ← in1 shift left logical in2	For SLL
1110	out ← in1 shift right logical in2	For SRL
1111	out ← in1 shift right arithmetic in2	For SRA

- The result of ==, ! =, < and >= becomes 32’b1 or 32’b0.
- The $\oplus$ denotes an exclusive logical sum (XOR)bit by bit. The operator is ^ in Verilog-HDL.
- The $\vee$ denotes a logical sum (OR) bit by bit. The operator is | in Verilog-HDL.
- The $\wedge$ denotes a logical conjunction (AND) bit by bit. The operator is & in Verilog-HDL.
- The ‘shift left logical’ left-shifts the in1’s value by the in2’ value. 0 is entered to the lowest bit.

- The ‘shift right logical’ right-shifts the in1’s value by the in2’ value. 0 is entered to the lowest bit.

- The ‘shift arithmetic’ right-shifts the in1’s value by the in2’ value, while it doesn’t change the uppermost bit keeping the original value. This is the ingenuity that the right-shift operation can become equivalent to dividing by 2, when regarding the original value as a signed integer in the form of 2’s complement. For that, this shift is called arithmetic.

This is described in the public/alu.v.

C.9.6 CSR

Template of CSR

```
module csr(input                    clock,    // Clock
input                     reset,    // Reset
input [11:0]              addr,     // Address
input [31:0]              in,       // Input
input [1:0]               op,       // Operation instruction
output [31:0]             out,      // Output

input                     mie_in    // Input to MSTATUS.MIE
input                     mie_ld    // Write control signal of MSTATUS.MIE
output                    mie_out // Output of MSTATUS.MIE

input                     mpie_in // Input to MSTATUS.MPIE
input                     mpie_ld // Write-in control signal to MSTATUS.MPIE
output              mpie_out // Output of MSTATUS.MPIE

input                     mtie_in   // Input to MIE.MTIE
input                     mtie_ld   // Write-in control signal to MIE.MTIE
output              mtie_out // Output of MIE.MTIE

input [31:0]              mepc_in // Input to MEPC
input                     mepc_ld // Write-in control signal to MEPC
output [31:0]             mepc_out // Output of MEPC

input [31:0]              mcause_in // Input to MCAUSE
input                     mcause_ld // Write-in control signal to MCAUSE
output [31:0]             mcause_out // Output of MCAUSE

input                     mtip_in // Input to MIP.MTIP
input                     mtip_ld // Write-in control signal to MIP.MTIP
output              mtip_out // Output of MIP.MTIP
);
...
endmodule
```

The CSR is a kind of register files and an individual file has a special meaning. Then, Inaddition to the interface used in instruction execution, the interface individually referred to in the interrupt procedure is also prepared. Specifically, the following signal line is used during the execution of the CSR instruction.

- addr Designate the address of the CSR register.

- in An input value in the CSR instruction.

- op Type of the CSR instruction. The following codings are to be used here.

Table C.14 Encoding of op

op	Meaning
00	Nop. Nothing done.
01	write. Write the value of ‘in’, as it is.
10	set. Take the logica sum with the value of ‘in’.
11	clear. Take the logical conjunction with the value that the ‘in’ is inversed in the unit of bit.

- out The register’s value designated by the CSR instruction.

Usually the in’ is the value of the register designated by *R_{s1}*. However, in the case of the CSR instruction related to the immediate value like the CSRRWI, mind that what 0 is entered to the uppermost 27 bits is used regarding the value of *R_{s1}* as an immediate value of 5 bits, though it is not what the CSR module performs. Depending on the value of ‘op’, the logical operation with ‘in’ is needed. Because this is a simple operation of AND and OR, it is processed in the module of the CSR, not using the ALU.

Other signal lines are composed the XXX_in, XXX_ld and XXX_out. The XXX_in is for input, the XXX_ld is for write-control-signal and the XXX_out is for output, respectively. These are directly connected to the input of corresponding register, the write-control-signal and the output. Notice that each register of the CSR may be accessed directly related to the hardware interrupt, other than the instructions of the CSR group.

The KAPPA3-LIGHT doesn’t use the CSR.

C.9.7 Phase generator

Tempate for phase generator

```
module phasegen(input          clock,          // Clock
                input          reset,          // Reset
                input          run,            // run signal
                input          step_phase,     // Signal to execute in the unit of phase
                input          step_inst,      //Signal ro execute in the unit of n instructions
                output [3:0]   cstate,         // Phase output
                output         running);       // Signal showing 'program running'
...
endmodule
```

The phase generator is, to be exact, a part of controller. But in the KAPPA3-LIGHT, the module for the phase transition is made independent so as to have a function to stop executing at every phase or every instruction. As a result, it has become a pure combinational circuit that doesn't have a memory within a controller.

The cstate is a signal line of 4 bits expressing the phase and the coding for the cstate should be performed as follows.

Table C.15 Coding of cstate

Phase	cstate
IF	4'b0001
DE	4'b0010
EX	4'b0100
WB	4'b1000

As far as the phase generator is working normally, the cstate shouldn't take any values other than four ones above. However, it is desirable to transit to the IF phase at the next clock in the case of illegal value, for security.

Configuration It has the register retaining the value of a phase signal of 4 bits (Each phase shoud be reg-declared.) and the register retaining an internal state as follows. Mind that the internal state referred here is irrevalent to the phase. Because there are four internal states, the register of 2 bits is necessary to retain these 4 states, at minimum. It is free how each state is assigned to one of the codes.

Table C.16 Internal state of phase generator

Internal state	Meaning
STOP	Stopage state
RUN	Ordinary execution mode
STEP INST	Mode that stops at every instruction
STEP PHASE	Mode that stops at every phase

Operation **The process is carried out, following priority below.**

- When the reset is 0, set the phase ‘IF’ and set the internal state ‘Stop’.

- Following the internal state, perform the operations below.

STOP - When the run is 1, go to ‘RUN’.

-When the step_inst is 1, go to ‘STEP_INST’.

- When the step_phase is 1, go to ‘STEP_PHASE’.

- Otherwise, the state doesn’t change.

RUN - When the run is 1, set the next state ‘STOP’.

- Otherwise, advance the phase by one. The next state is kept ‘RUN’.

STEP_INST - When the phase is WB, set the phase ‘IF’ and set the state ‘STOP’.

- Otherwise, advance the phase by one. The next state is kept ‘STATE_INST’.

STEP_PHASE - Advance the phase by 1.

The next state should be ‘STOP’ without fail.

Note that the signal, ‘running’ becomes 1 when the internal state is not ‘STOP’. Create as a pure combinational circuit.

C.9.8 Controller

Template of controller

```
module controller(input [3:0]       cstate,         // Phase signal
                  input [31:0]      ir,             // IR register's valeu
                  input [31:0]      addr,           // Memory address
                  input [31:0]      alu_out,        // ALU's output

                  output            pc_sel,         // PC's input selection
                  output            pc_ld,          // PC's write-control
                  output            mem_sel,        // Input selection of memory address
                  output            mem_read,       // Memory's read-control
                  output            mem_write,      // Memory's write-control
                  output [3:0]      mem_wrbits,     // Memory's write-bit-mask
                  output            ir_ld,          // IR register's write-control
                  output [4:0]      rs1_addr,         // RS1 address
                  output [4:0]      rs2_addr,       // RS2 address
                  output [4:0]      rd_addr,        // RD address
                  output [1:0]      rd_sel,         // RD's input selection
                  output            rd_ld,          // RD's write-control
                  output            a_ld,           // A register's write-control
                  output            b_ld,           // B register's write-control
                  output            a_sel,          // Input selection of ALU's input 1
                  output            b_sel,          // Input selection of ALU's input 2
                  output [31:0]     imm,            // Immediate value
                  output [3:0]      alu_ctl,        // ALU's function code
                  output            c_ld);          // C register's write-control
...
endmodule
```

Configuration **Design this module as a combinational circuit.**

Operation **The signal to control other modules is to be generated. The details are as follows.**

Cntrol of PC (pc_sel and pc_ld) The value of the PC changes in the cases below.

- **The cstste is 'IF'. This case, always add 4.**
- **JAL and JALR instruction, and the cstate is 'WB'. This case, substitute the C register's value.**
- **Conditioned branch instruction like BEQ instruction, and the cstate is 'BEQ'. This case, substitute the C register's value only when the branch condition is satisfied. The alu_out has the result of branch condition.**

Control of memory (mem_sel, mem_read, mem_write and mem_wrbits)

The memory address is designated in the following cases.
- The cstate is 'IF'. This case, select the PC's value.
- The load and store instruction, and the cstate is 'WB'. This case, select the C register's value.
The content of memory is read out in the following cases.
- The load instruction, and the cstate is 'WB'.
The content of memory changes in the following cases.
- The store instruction, and the cstate is 'WB'.
Set the bit mask to write into memory as follows. Though the memory is basically accessed in the unit of one word (32 bits = 4 bytes) in RV32I, the sb instruction and sh instruction are needed to be accessed in the unit of 8 bits = 1 byte and 16 bits = 2 bytes, respectively. For that, the signal line of mem_wrbits is prepared for designating the byte to be rewritten in the 4 bytes. This is the signal line of 4 bits and each bit corresponds to the 0th to 3rd bit. For example, in the case of writing the 0th bit only, designate 4'b0001. In the case of writing all of the 4 bytes, use 4'b1111.
- In the case of the sb instruction, only one of 0th, 1st, 2nd and 3rd bit is set 1, according to the lower 3 bits of memory address.
- In the case of the sh instruction, set 1 to the bit of either the 0th and 1st byte or the 2nd and 3rd byte, depending on whether the lower 2 bits of memory address is 2'00 or 2'10.
- In the case of the sw instruction, all of the bytes are to be written, so it becomes 4'b1111.

Control of IR (ir_ld) The content of the IR is changed in the following case.
- The cstate is 'IF'.

Control of register file Part 1 (rs1_addr, rs2_addr and rd_addr) As the fields of *rs*1, *rs*2 a nd *rd* are the same for all the instruction formats in the RV32I, use that for rs1_addr, rs2_addr and rd_addr, as it is.

Control of register file Part 2 (a_ld and b_ld) The content of the register file is read out to the A register and B register in the following case.
- The cstate is 'DE'.
There are cases not using the values of the A register and the B register depending on the instructions. That case, this reading process becomes waisted. However, the creation of the logical circuit for conditional judgement is more waisting, so the unconditional read-out makes the logical circuit simpler.

Control of register file Part 3 (rd_sel and rd_ld) The content of register file changes in the following cases.

- The cstate is 'WB' in the operation instruction. This case, the C register's value is written.
- The cstate is 'WB' in the load instruction. This case, the value of memory's output is written.
- The cstate is 'WB' in the jump instruction. This case, the PC register's value is written.

Refer to the phase table for the details.

Creation of immediate value (imm) There are the following types of immediate values, depending on the instruction formats.

- I_imm : The immediate value used for the I-type instruction format. This sign-extends the 12 bits signed integer to the 32 bits signed integer.
- S_imm : The immediate value used for the S-type instruction format. This sign-extends the 12 bits signed integer to the 32 bits signed integer. The difference from the I_imm is the bit position generating the immediate value.
- B_imm : The immediate value used for the B-type instruction format. Because the address of branch destination is not an odd number, the 0th bit is always 0. In the IR, designate 12 bits from 1st to 12th bit. After that, sign-extend to the signed integer of 32 bits. As this format is complicated, confirm it carefully.
- U_imm : The immediate value used for the U-type instruction format. About the 20 bits designated in the IR, set 0 to the lower 12 bits, using the upper 20 bits.
- J_imm : The immediate value used for the J-type instruction format. Because the address of branch destination is not an odd number, the 0th bit is always 0. In the IR, designate 20 bits from 1st to 20th bit. After that, sign-extend to the signed integer of 32 bits. As this format is complicated, confirm it carefully.
- Shift instruction of immediate value : This is roughly classified into the I-type instruction. Because the shift amount is 32 bits at maximum in the operation of 32 bits, it is used only the lower 5 bits in the field of the I-type immediate value. The upper bits are used to distinguish between the srli instruction and srla instruction.

Control of ALU (a_sel, b_sel and alu_ctl) When the cstate is 'EX', the ALU is used in some form or other. (Refer to the phase table C.8), so this outputs the function codes to the alu_ctl, depending on the content. At the same time, set appriate values to the a_sel and b_sel. Further, in conditioned branch instruction, the ALU is used by the judgement for branch condition, even when the cstate is 'WB'.

Control of C register (c_ld) The content of the C register changes in the following case.

- The cstate is 'E'.

C.9.9 Memory

Memory

```
module memory(input          clock,          // Clock
              input [31:0]   address,        // Address
              input          read,           // Read-out enable
              input          write,          // Write-in enable
              input [31:0]   wrdata,         // Write-data
              input [3:0]    wrbits,         // Write-bit-mask
              output [31:0]  rddata,         // Read-data

              input          dbg_mode,       // Debugging mode
              input [31:0]   dbg_address,    // Address for debugging
              input          dbg_read,       // Read-out enable for debugginb
              input          dbg_write,      // Write-in enable
              input [31:0]   dbg_in,         // Write-data for debugging
              output [31:0]  dbg_out);       // Read-data for debugging
...
endmodule
```

This is described in the public/memory.v. The public/memory.v is instantiated internally. The description on the mem64kd.v is a special one only for the Quartus.

C.9.10 KAPPA3-LIGHT core

KAPPA3LIGHT core

```
module kappa3_light_core(input c          lock,       // Clock
                         input            clock2,     // 2-divided clock
                         input            reset,      // Reset
                         input            run,        // 'run' signal
                         input            step_phase,      // 'SP' signal
                         input            step_inst,       // 'SI' signal

                         input [31:0]     dbg_in,          // Write-data for debug
                         input            dbg_pc_ld,   // Write-enable signal for PC debug
                         input            dbg_ir_ld,    //Wright-enable signal for IR debug
                         input            dbg_reg_ld,//Write-enable signal for REGFILE debug
                         input [4:0]      dbg_reg_addr,     // Address for REGFILE debugging
                         input            dbg_a_ld,   //Write-enable signal for A register debug
                         input            dbg_b_ld,   // Write-enable signal for B register debug
                         input            dbg_c_ld,   // Write-enable signal for C register debug
                         input [31:0]     dbg_mem_addr,  // Memory address for debug
                         input            dbg_mem_read, //Memory read-out signal for debug
                         input            dbg_mem_write, // Memorywrite-in signal for debug
                         output [31:0]    dbg_pc_out,         // PC's debug output
                         output [31:0]    dbg_ir_out,         // IR's debug output
                         output [31:0]    dbg_reg_out,        // REGFILE's debug output
                         output [31:0]    dbg_a_out,          // A regidter's debug output
                         output [31:0]    dbg_b_out,          // B register's debug output
                         output [31:0]    dbg_c_out,          // C register's debug output
                         output [31:0]    dbg_mem_out       // Read-out data from memory
...
endmodule
```

This is almost the same configuration as Figure C.2. It is added the interface signal (with the head, 'dbg') for debugging. The dbg_in is used as a value to be written, in common. The target to be written is designated by 'dbg_xx_ld. The register's value is taken from the 'dbg_xx_out. However, only for memory, the 'dbg_mem_read' is used for the read-out and the 'dbg_mem_write' is used for the write-in. The lower modules are the ALU, register file, PC, IR, A, B, C, STCONV, LDCONV, phase generator and controller. Among them, the PC, IR, A, B and C use the general register of 32 bits,

‘reg32’. The CSR is not included.

The public/kappa3 light core dp.v is the description instantiating the PC, IR, A, B, C, register file and memory, among them. Because there are no ALU, STCONV, LDCONV, phase generator and controller, this is not operated as a processor. But it can be used for the confirmation of debugger’s operation, as it has all of internal memory elements.

Only the memory uses the clock, and the remainings use the clock2. This is because the timing of reading/writing memory is taken into account.

C.9.11 Top module for KAPPA3-LIGHT

Top module for KAPPA3-LIGHT

```
module kappa3_light(input           sys_clock,      // System clock
                    input           reset,          // Reset
                    input           clock,          // CPU clock
                    input           psw_a0,         // Push SW-A0
                    input           psw_a1,         // Push SW-A1
                    input           psw_a2,         // Push SW-A2
                    input           psw_a3,         // Push SW-A3
                    input           psw_a4,         // Push SW-A4
                    input           psw_b0,         // Push SW-B0
                    input           psw_b1,         // Push SW-B1
                    input           psw_b2,         // Push SW-B2
                    input           psw_b3,         // Push SW-B3
                    input           psw_b4,         // Push SW-B4
                    input           psw_c0,         // Push SW-C0
                    input           psw_c1,         // Push SW-C1
                    input           psw_c2,         // Push SW-C2
                    input           psw_c3,         // Push SW-C3
                    input           psw_c4,         // Push SW-C4
                    input           psw_d0,         // Push SW-D0
                    input           psw_d1,         // Push SW-D1
                    input           psw_d2,         // Push SW-D2
                    input           psw_d3,         // Push SW-D3
                    input           psw_d4,         // Push SW-D4
                    input [3:0]     hex_a,          // Rotary SW HEX_A
                    input [3:0]     hex_b,          // Rotsry SW HEX_B
                    input [7:0]     dip_a,          // DIP-SW DIP_A
                    input [7:0]     dip_b,          // DIP-SW DIP_B
                    output [7:0]    seg_x,          // MU500-RK's board output (SEG_X)
                    output [3:0]    sel_x,          // MU500-RK's board output (SEL_X)
                    output [7:0]    seg_y,          // MU500-RK's board output (SEG_Y)
                    output [3:0]    sel_y,          // MU500-RK's board output (SEL_Y)
                    output [7:0]    led_out,        // MU500-RK's board output (LED_OUT)
                    output [7:0]    seg_a,          // MU500-7SEG's board output (SEG_A)
```

Continued to the next page

Continuation from the previous page to the next page

```
                    output [7:0]   seg_b,          // MU500-7SEG's board output (SEG_B)
                    output [7:0]   seg_c,          // MU500-7SEG's board output (SEG_C)
                    output [7:0]   seg_d,          // MU500-7SEG's board output (SEG_D)
                    output [7:0]   seg_e,          // MU500-7SEG's board output (SEG_E)
                    output [7:0]   seg_f,          // MU500-7SEG's board output (SEG_F)
                    output [7:0]   seg_g,          // MU500-7SEG's board output SEG_G)
                    output [7:0]   seg_h,          // MU500-7SEG's board output (SEG_H)
                    output [8:0]   sel);           // MU500-7SEG's board output (SEL)
...
```

This connects the KAPPA3-LIGHT core with external inputs/outouts. Use the description of the public/kappa3 light.v.

References

[1] Mitsubishi Electric Micro-Computer Application Software Co.,Ltd, "MU500-RX Set Users Manual Ver 1.1.pdf".

[2] Mitsubishi Electric Micro-Computer Application Software Co.,Ltd, "MU500-7SEG Manual Ver 2.pdf".

[3] Mitsubishi Electric Micro-Computer Application Software Co.,Ltd, "FPGA Design Tool Operating Procedure Manual (RX dedicated for Quartus II) v122.pdf"

[4] Shinya Kimura, "Verilog-HDL Logical Circuit Design", CQ Publishing, Co., Ltd., 2001.